UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

Ambient Occlusion on Mobile: An Empirical Comparison

Author:
Marc Sunet

Supervisors:
Pere-Pau Vázquez

A thesis submitted in fulfilment of the requirements
for the degree of Master in Innovation and Research in Informatics

in the

Facultat d'Informàtica de Barcelona
Department of Computer Science

January 25, 2016

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

Facultat d'Informàtica de Barcelona

Department of Computer Science

Master in Innovation and Research in Informatics

Ambient Occlusion on Mobile:
An Empirical Comparison

by Marc Sunet

Screen space ambient occlusion is a popular dynamic global illumination technique that has seen widespread adoption in PC computer games and other computer graphics applications due to its simplicity, scalability, and ability to be integrated with other techniques. Mobile platforms, however, have traditionally been unable to run screen space ambient occlusion and other global illumination techniques in real-time, forcing developers to bake most of the illumination as a consequence. On the other hand, the mobile devices are evolving very rapidly, and mobile GPUs deliver ever more stunning results with every generation. In this thesis, we study the feasibility of screen space ambient occlusion on a range of devices. We implement several of the most popular techniques and propose two rendering pipelines to support them, as well a mobile-friendly algorithm to approximate ambient occlusion and a modification that can be applied on any of the techniques to speed up their computation times. Our findings suggest that screen space ambient occlusion is indeed feasible on middle- to high-end devices, with frame rates ranging from 20 to 60+ frames per second depending on the algorithm and device used. Given the history and trend of the evolution of mobile GPUs, it seems to be only a matter of time before screen space ambient occlusion and other global illumination techniques become standard in the mobile domain.

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

Facultat d'Informàtica de Barcelona

Department of Computer Science

Master in Innovation and Research in Informatics

Ambient Occlusion on Mobile:
An Empirical Comparison

by Marc Sunet

Screen space ambient occlusion és una popular tècnica d'il·luminació global dinàmica que ha sigut adoptada per una gran quantitat de jocs PC i altres aplicacions de gràfics per computador degut a la seva senzillesa, escalabilitat i habilitat de ser integrada amb altres tècniques. Les plataformes mòbils, no obstant, han sigut tradicionalment incapaços de calcular screen space ambient occlusion i altres tècniques d'il·luminació global en temps real, forçant als desenvolupadors a pre-calcular gran part de la il·luminació com a conseqüència. Per altra banda, els dispositius mòbils estan evolucionant molt ràpidament, i les GPUs mòbils se superen l'una a l'altra generació rere generació. En aquesta tesi, estudiem la factibilitat de la implementació de screen space ambient occlusion per a mòbils en una varietat de dispositius. Implementem algunes de les tècniques més populars i proposem dos pipelines de rendering per suportar-les, a més d'una tècnica òptima per a dispositius mòbils per aproximar la oclusió ambient i una modificació aplicable a totes lès tècniques per millorar el seu rendiment. Els nostres resultats suggereixen que la tècnica de screen space ambient occlusion és factible en dispositius de gamma mitja-alta, obtenint frame rates de 20-60+ frames per segon dependent de l'algorisme i dispositiu utilitzats. Donada la història i l'evolució de les GPUs mòbils, creiem que només es qüestió de temps que tècniques d'iluminació global com screen space ambient occlusion s'estandarditzin en l'espai mòbil.

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

Facultat d'Informàtica de Barcelona

Department of Computer Science

Master in Innovation and Research in Informatics

Ambient Occlusion on Mobile:
An Empirical Comparison

by Marc Sunet

Screen space ambient occlusion es una popular técnica de iluminación global dinámica que ha sido adoptada por una gran cantidad de juegos PC y otras aplicaciones de gráficos por computador debido a su sencillez, escalabilidad y habilidad de ser integrada con otras técnicas. Los dispositivos móviles, sin embargo, han sido tradicionalmente incapaces de calcular screen space ambient occlusion y otras técnicas de iluminación global en tiempo real, obligando a los desarrolladores a pre-calcular gran parte de la iluminación como consecuencia. Por otro lado, los dispositivos móviles están evolucionando muy rápidamente, y las GPUs móviles se superan una a la otra generación tras generación. En esta tesis, estudiamos la factiblidad de la implementación de screen space ambient occlusion para móviles en una variedad de dispositivos. Implementamos algunas de las técnicas más populares i proponemos dos pipelines de rendering para soportarlas, además de una técnica óptima para dispositivos móviles para aproximar la oclusión ambiente y una modificación aplicable a todas las técnicas para mejorar su rendimiento. Nuestros resultados sugieren que la técnica de screen space ambient occlusion es factible en dispositivos de gama media-alta, obteniendo frame rates de 20-60+ frames per segundo dependiendo del algoritmo y dispositivo utilizados. Dada la historia y la evolución de las GPUs móviles, creemos que sólo es cuestión de tiempo que técnicas de iluminación global como screen space ambient occlusion se estandaricen en el espacio móvil.

Acknowledgements

I would like to thank my supervisor Pere-Pau Vázquez and my girlfriend Nensi for their guidance and unconditional support throughout the development of this thesis.

Contents

List of Figures

Chapter 1

Introduction

Ambient occlusion (AO) is a shading technique that approximates global illumination by determining how exposed a point on a surface is to ambient lighting. Screen space ambient occlusion approximates the true ambient occlusion in a scene in a post-process rendering step. Since this form of ambient occlusion operates in screen space, it is independent from the scene's geometry and is therefore faster to compute. In addition, screen space ambient occlusion is relatively simple to implement, and can be integrated into a wide variety of computer graphics applications.

An application domain that has seen a widespread adoption of screen space ambient occlusion methods is that of 3D video games. Screen space ambient occlusion techniques can be computed in real-time using modern graphics cards, and often consume just a fraction of a game's time budget, leaving space for other stages of the rendering pipeline and the other subsystems involved in such applications. In addition, screen space ambient occlusion can be combined with other global illumination effects, often complementing them by adding high-frequency, subtle shading details in the scene. For these reasons, screen ambient occlusion remains a popular technique in today's games.

While screen space ambient occlusion is heavily deployed in computer titles, its use in the mobile space has been very limited. Though relatively powerful, most mobile graphics chips are still considerably behind desktop ones in both computational power and memory bandwidth, especially those found in the average low- to middle-end phones. Computing full global illumination effects in real-time and at reasonable framerates in these devices is expensive and unpractical, and as a consequence, game engines often bake a game's illumination when targeting mobile platforms to provide the gamer with a smooth experience. Techniques such as screen space ambient occlusion are therefore rarely used in the mobile domain.

Although still behind desktop GPUs, mobile graphics chips are evolving at an unrested pace. The Adreno 330 GPU, powering Google's Nexus 5 (October 2013), is a fully compatible OpenGL ES 3.0 device featuring 86-97 GFLOPS[1]. The more recent Adreno 420,

[1] https://en.wikipedia.org/wiki/Adreno

on the other hand, is a 3.1 compatible device found in Google's Nexus 6 (October 2014) and offering 144-172 GFLOPS. On the higher end, NVIDIA's Tegra K1, powering the NVIDIA Shield K1 (December 2015), features 192 CUDA cores at 365 GFLOPS[2]. Judging by the dates and numbers and the high demand of mobile devices in the market, we believe it is only a matter of time that these middle- to high-end GPUs take over and that low-end ones are phased out. In fact, according to Unity's hardware statistics page, ES 3.0 compatible devices already make up 44.9% of the user base at the time of writing[3].

The purpose of this thesis is to evaluate and study the performance and runtime characteristics of different screen space ambient occlusion techniques on middle to high-end mobile devices. We implement several of the most popular techniques and modify them to better adapt them to mobile. We benchmark them and compare their performance. We identify bottlenecks in the rendering pipeline and optimise them to develop a usable framework in which different ambient occlusion techniques can be implemented.

In chapter 2, we introduce ambient occlusion and the mathematical foundations on which it is based. We then present a brief overview of different real-time ambient occlusion methods, including screen space ambient occlusion, and end the chapter with a discussion on mobile GPU architecture that will be helpful in future chapters.

Chapter 3 is exclusively devoted to screen space ambient occlusion. In the first half, we continue from the brief introduction in the previous chapter and discuss screen space techniques in greater detail. In the second half, we present the previous work our study is based on.

In chapter 4, we cover our implementation of screen space ambient occlusion for mobile. We discuss our rendering pipeline and our implementation of the different ambient occlusion techniques presented in the previous chapter. For convenience, we have written this chapter so that it can be used as a reference. That is, this chapter only explains the *how*; the *why* is deferred to the following chapter.

Chapter 5 shows our results. Here, we show the different experiments we performed during the development of our project and justify the decisions taken. This chapter complements the previous one by providing the *why* in our implementation. The chapter concludes with a performance and a qualitative comparison of the ambient occlusion techniques we have implemented.

Finally, in chapter 6, we state our conclusions and discuss potential future work.

[2] http://www.anandtech.com/show/7622/nvidia-tegra-k1/3
[3] http://hwstats.unity3d.com/mobile/gpu.html

Chapter 2

Background

This chapter introduces the necessary background needed to follow the rest of this document. An intuitive description of ambient occlusion is first presented, followed by its mathematical derivation and an overview of the different real-time methods ambient occlusion methods that are in use today.

2.1 Introduction to Ambient Occlusion

Ambient occlusion is a shading technique that shades points as a function of their visibility. Points that are occluded by nearby geometry are shaded in a dark shade of gray, whereas points that are relatively unoccluded appear in a lighter shade of gray. An example of a scene rendered with ambient occlusion can be seen in figure 2.1.

FIGURE 2.1: Ambient occlusion in an example scene.
Source: Game Informer.

The advantage of using ambient occlusion is best understood when comparing a scene with and without ambient occlusion side by side. Figure 2.2 presents one such comparison. On the left, the scene is rendered using a constant ambient term, which is very common in lighting models such as Blinn-Phong or Cook-Torrance. As a consequence, the object appears flat, making it impossible for the viewer to appreciate the

object's 3D features. In contrast, these features are clearly visible in the right figure, which shows the same object rendered with ambient occlusion. Since ambient occlusion shades occluded points darker than unoccluded points, cracks and crevices appear darker, revealing the object's features and depth complexity.

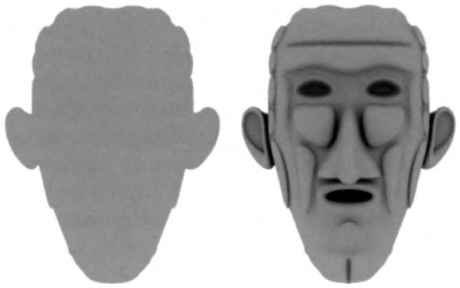


FIGURE 2.2: Ambient occlusion versus a constant ambient term.
Source: PRTDemo, Microsoft SDK, November 2007

At this point, one could think of ambient occlusion as a depth cue or depth-enhancing technique, and this is in fact an intuitive way to understand it. Ambient occlusion is not the first depth-enhancing technique, however, as earlier methods had been proposed. [TL06] presents a technique that produces an unsharp mask of the depth buffer to enhance depth discontinuities, resulting in a darkening of these points in the image and making it easier for the viewer to tell objects apart and to better understand the depth complexity of individual objects. This technique is illustrated in figure 4.21. [Rit+08] is a similar technique that also enhances depth discontinuities to make object features more prominent, as illustrated in figure 2.4.

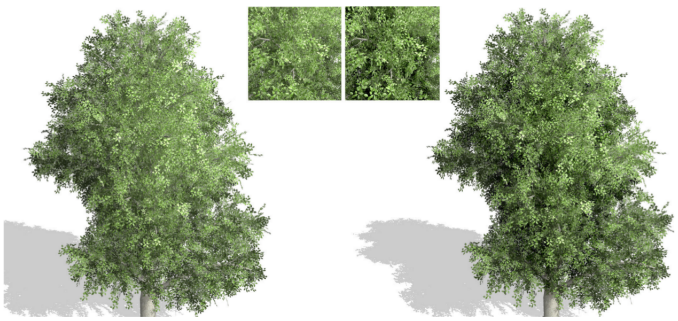


FIGURE 2.3: Enhancement of a complex botanical object using depth darkening [TL06].

What sets ambient occlusion apart from those techniques is that ambient occlusion is based on the physical behaviour of light. For this reason, and although an intuitive understanding is enough for many practical purposes, it is helpful to understand the mathematical derivation of ambient occlusion. This will give us further insight into the concept and will allow us to better understand and compare different ambient occlusion techniques.

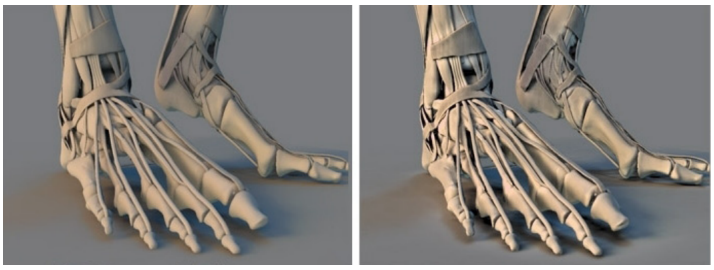


FIGURE 2.4: A scene with and without 3D unsharp masking enhancement [Rit+08].

2.2 The Rendering Equation

To understand the mathematical foundations of ambient occlusion, we first need a basic understanding of how light behaves and how objects reflect light.

The interaction between light and objects is modelled by what is known as the rendering equation. In other words, the rendering equation describes how light reflects off a surface. Informally, we could say that if the rendering equation were to be fully solved for every point in a scene, one would produce images that would be indistinguishable from real life. Some computer graphics techniques approximate the rendering equation with great detail, producing stunning images such as the one in figure 2.5. In this sense, we can think of the rendering equation as the superset of all computer graphics techniques. Or in other words, every computer graphics technique is a subset of the effects described by the rendering equation.

FIGURE 2.5: Path tracing with Brigade 3.
Source: http://www.evermotion.org.

The rendering equation does not have an analytical solution, however, so one can only approximate it in practise. The degree to which the rendering equation is approximated determines how realistic the final rendering is, with better approximations producing more realistic images. In this sense, every computer graphics technique can be understood as an approximation of the rendering equation, and ambient occlusion is no exception. Understanding the rendering equation, then, is essential to fully understand computer graphics techniques such as ambient occlusion.

The rendering equation can be derived in an intuitive and straightforward manner that is enough for our purposes. Consider the scene in figure 2.6. Here, the viewer sees a point p on a given surface, and the goal of a rendering system is to compute the light that is reflected off of p in the outgoing direction ω_o towards the viewer. The light reflected off of p and towards the viewer is a function of two components: the amount of light incident at p from every incoming direction ω_i and the properties of the surface, which describe how the incoming light is reflected.

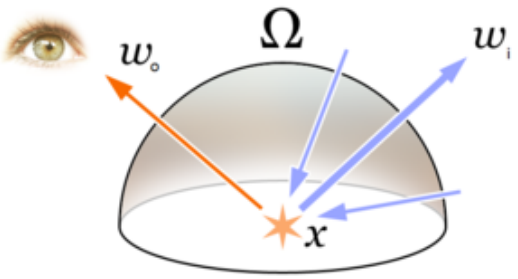


FIGURE 2.6: Light reflected at a surface point p. Source: Wikipedia.

The surface properties are often described with what is known as a *bidirectional reflectance distribution function*, or BRDF for short. The BRDF can be understood as the answer to a simple question: how much light is reflected off a point in one direction due to light incoming in another direction? This is illustrated in figure 2.7. The BRDF is therefore a function of three parameters: the surface point p, the incoming light direction ω_i, and the outgoing light direction ω_o, and is commonly denoted $f(p, \omega_o, \omega_i)$.

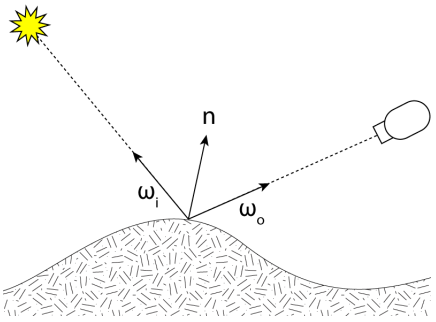


FIGURE 2.7: The BRDF describes how much light is reflected off a point in one direction due to light incoming in another direction. Source: Wikipedia.

If we take every beam of light incident at p in direction ω_i, multiply this quantity by the BRDF $f(p, \omega_o, \omega_i)$ and add everything up, we obtain the light reflected at p in direction ω_o. This process is exactly what is described by the rendering equation:

$$L_o(p, \omega_o) = \int_\Omega f(p, \omega_o, \omega_i)\, L_i(p, \omega_i)\, \cos\theta_i \, \mathrm{d}\omega_i \tag{2.1}$$

In the above equation, $L_o(p, \omega_o)$ denotes the light reflected at p in direction ω_o. This quantity is essentially the sum of the incident light $L_i(p, \omega_i)$ in direction ω_i multiplied by the BRDF $f(p, \omega_o, \omega_i)$, for every ω_i in the normal-oriented hemisphere, where ω_i is a differential solid angle in this hemisphere (a infinitesimally thin cone of incident directions). This product is attenuated by $\cos\theta$, which is the cosine of the angle between the surface normal and the direction of incoming light ω_i. This cosine factor is due to Lambert's cosine law, which essentially states that light striking the surface at normal incidence has a stronger influence on the reflected light than light coming at grazing angles, and this effect is modelled by taking the cosine of the angle.

Careful observation of the above equation reveals a subtle problem when it comes to computing the light reflected off a surface: L_o is a function of L_i as expressed by the equation, but L_i is also in fact a function of L_o: some of the light that is reflected off a surface bounces around the scene and eventually hits back that same surface. This is essentially where the complexity of the equation lies, and in practise, rendering systems approximate these light bounces using different methods to achieve different results. Methods that result in better approximations generally result in higher visual fidelity, but are more costly to compute. A rendering system must find a balance between visual quality and performance.

2.3 From the Rendering Equation to Ambient Occlusion

Ambient occlusion can be understood as an approximation to the rendering equation. Let us see how to derive the mathematical definition of ambient occlusion, starting with a simple approximation and adding more complexity to reach our goal.

2.3.1 Direct Lighting

Perhaps the simplest approach when it comes to approximating the rendering equation (conceptually) is to simply ignore indirect lighting, or light bouncing off from one surface to another, and to compute direct lighting only, or light coming directly from a light source. This produces images like the one in figure 2.8, where only points that are visible from the light source are shaded, and those from which the light source is not reachable appear pitch black.

This method is a very crude approximation of the rendering equation, and the visual result is unacceptable for most practical purposes. In practise, we need to approximate indirect lighting for results to be visually plausible, as illustrated in figure 2.9. This figure shows the same scene rendered with 0, 1, and 2 bounces of indirect light. The leftmost image is the same as in figure 2.8: zero bounces of light correspond to direct lighting. The image in the middle adds one bounce of indirect lighting. Notice

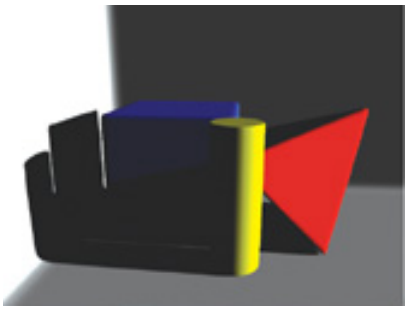


FIGURE 2.8: An example scene using direct lighting only. Source: [PF05].

how points in shadow are no longer pitch black, but instead receive illumination from nearby objects. The rightmost image adds another bounce of light, for a total of two bounces. The difference between the middle and rightmost image is subtle and hard to appreciate on print, but in the rightmost image, the shadow cast by the blue box onto the wall takes a stronger shade of blue and appears brighter.

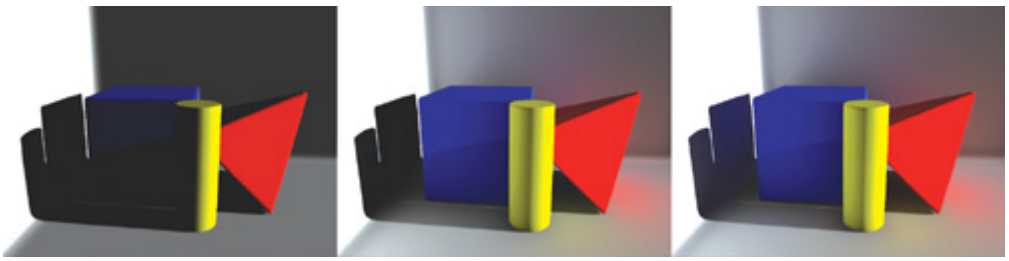


FIGURE 2.9: An example scene using 0, 1, and 2 bounces of lighting. Source: [PF05].

Adding an increasingly number of bounces produces a better approximation of the rendering equation, but becomes computationally more expensive. Even though modern rendering systems can compute one or two bounces of indirect light in real time, these systems had traditionally offered approximations to the rendering equation that were cheaper to compute and offered visually plausible results.

2.3.2 Ambient Light

A traditional and computationally-efficient, albeit rough method of approximating indirect illumination is to replace the computation altogether with a constant. This constant term is referred to as *ambient light* in shading models such as Blinn-Phong or Cook-Torrance. By using a constant, we assume that light reaches every point in every direction and with equal intensity, ignoring the point's surrounding objects. This results in images such as the one in figure 2.10.

In figure 2.10, we can see how all points receive some light, be it from the light source or from the artificial ambient light term. In contrast to figure 2.8, no point in this image appears pitch black, rendering the image more visually appealing. Nevertheless, a

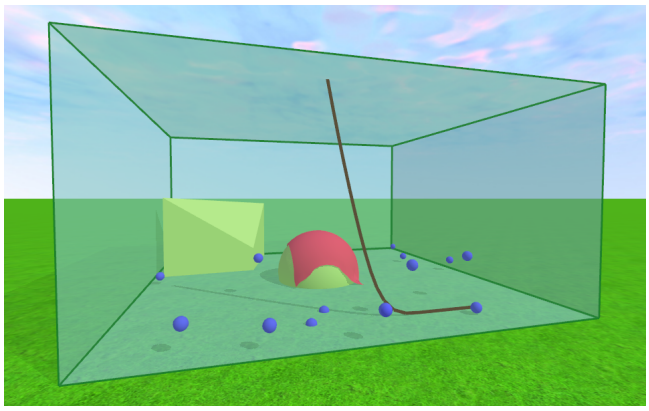


FIGURE 2.10: A scene illuminated using ambient light.

constant ambient term makes the scene appear dull and uninteresting, and it makes it difficult for the viewer to appreciate the depth complexity of the scene. This is the exact same problem ambient occlusion tries to solve.

2.3.3 Ambient Occlusion

Ambient occlusion can be seen as a form of indirect lighting that is halfway between ambient light and true indirect illumination such as the one in figure 2.9. Instead of using a constant ambient term, ambient occlusion produces an ambient term for every point in the scene. This position-dependent value is a function of the point's visibility, and ambient occlusion makes two assumptions when computing this value.

The first assumption is that the surface is a perfect diffuse surface, that is, the surface reflects light equally in all directions. In this case, the surface's BRDF $f(p, \omega_o, \omega_i)$ becomes a constant k, in which case it can be moved out of the integral in equation 2.1 to yield the following expression:

$$L_o(p, \omega_o) = k \int_\Omega L_i(p, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i \tag{2.2}$$

The second assumption is that light potentially reaches a point p equally in all directions. In other words, the direction in which light actually reaches p becomes irrelevant, and we assume that the intensity of light potentially reaching p is equal in all directions. For this reason, ambient occlusion is said to be *undirectional*.

Notice the use of the word *potentially* above. Ambient occlusion does not simply add the light contribution from all directions, as that would result in a constant ambient light term. Instead, ambient occlusion takes into account the point's visibility. For every direction ω_i in which light may potentially reach p, ambient occlusion determines whether a ray in direction ω_i hits another surface. If it does, the point p is said to be

occluded in direction ω_i, in which case no light contribution in direction ω_i is added. On the other hand, if the ray does not intersect a surface, light is determined to reach p in direction ω_i, in which case this light contributes to the final value of reflected light L_o. This verbose description is more succinctly expressed by the following equation:

$$L_o(p, \omega_o) = k \int_\Omega V(p, \omega_i)\ \cos\theta_i \, \mathrm{d}\omega_i \tag{2.3}$$

where

$$V(p, \omega_i) = \begin{cases} 0 & p \text{ occluded in direction } \omega_i \\ 1 & \text{otherwise} \end{cases}$$

The difference between equations 2.3 and 2.2 is that the incoming light term $L_i(p, \omega_i)$ in equation 2.2 has been replaced by a visibility function $V(p, \omega_i)$. This visibility function returns 0 if p is occluded in direction ω_i and 1 otherwise. In this way, ambient occlusion adds the light contribution in direction ω_i only if p is unoccluded in this direction (there is no other surface blocking light in direction ω_i).

Finally, we derive the value of the constant k to arrive to the final ambient occlusion equation. For convenience, we define ambient occlusion to be a value in the range $[0, 1]$. For a completely unoccluded point, the visibility function $V(p, \omega_i)$ is 1 for every ω_i, therefore:

$$\begin{aligned} L_o(p_{\text{unoccluded}}, \omega_o) &= k \int_\Omega V(p_{\text{unoccluded}}, \omega_i)\ \cos\theta_i \, \mathrm{d}\omega_i \\ &= k \int_\Omega \cos\theta_i \, \mathrm{d}\omega_i \\ &= k\pi \end{aligned}$$

We want the ambient occlusion value for an unoccluded point to be 1, therefore

$$k = \frac{1}{\pi}$$

and the final ambient occlusion equation becomes:

$$L_o(p, \omega_o) = \frac{1}{\pi} \int_\Omega V(p, \omega_i)\ \cos\theta_i \, \mathrm{d}\omega_i \tag{2.4}$$

2.3.4 Ambient Obscurance

The above definition for ambient occlusion reveals a subtle problem. What is the value of ambient occlusion for a closed scene? The answer is zero, or pitch black. This is because every ray shot from every point p eventually hits another surface, in which case $V(p, \omega_i)$ is zero for all p and for all ω_i.

The above problem has a very simple solution. Instead of using the visibility function $V(p, \omega_i)$, we compute the distance from p to the potential occluder in direction ω_i, denoted $d(p, \omega_i)$, and then apply a *fall-off* function ρ to this distance to yield an occlusion value:

$$L_o(p, \omega_o) = \frac{1}{\pi} \int_\Omega \rho(d(p, \omega_i)) \cos\theta_i \, \mathrm{d}\omega_i \tag{2.5}$$

This modified equation is the *ambient obscurance* equation. Ambient obscurance can be therefore understood as a distance-limited form of ambient occlusion.

In practise, the fall-off function ρ in equation 2.5 typically cuts the occlusion value to zero when the distance is greater than a certain threshold, that is:

$$\rho(d) = \begin{cases} f(d) \in [0, 1] & d < \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

The fall-off function ρ therefore serves three purposes. First, ρ makes ambient obscurance work for closed scenes by yielding 0 for occluders that lie at a distance greater than some threshold — the obscurance value is not pitch black in closed scenes unlike ambient occlusion. Second, since ρ cuts the occlusion value to zero for far away points, the ambient obscurance of a given point becomes a relatively local computation. That is to say, the ambient obscurance value is only a function of the point's nearby geometry, which in turn translates to more efficient implementations in practise. Finally, ρ gives nearby occluders a greater occlusion value (a value closer to 0) than distant occluders.

Finally, it is important to note that in the literature, the terms *ambient occlusion* and *ambient obscurance* are often used interchangeably to refer to the latter. This is because ambient occlusion is of limited use in practise (since it is pitch black for closed scenes), so the term *ambient occlusion* is used even though what is really being referred to is ambient obscurance. We will follow this same convention for convenience, and use the term *ambient occlusion* as a synonym for ambient obscurance throughout the rest of this document.

2.4 Real-Time Ambient Occlusion Methods

As just seen, ambient occlusion is defined as an integral over the normal-oriented hemisphere. Like the rendering equation, the ambient occlusion equation does not have a general analytical solution, so it must be approximated in practise. The most often used method to approximate this equation is Monte Carlo integration. In Monte Carlo integration, an integral is approximated by randomly sampling the integrand and computing the average value of the individual results. Although ambient occlusion methods differ in implementation, they all share this scheme and define three key ingredients:

- A 3D representation of the scene.
- A sampling scheme.
- A fall-off function.

A 3D representation of the scene is needed to compute ray-object intersections. A sampling scheme defines how the normal-oriented hemisphere is sampled to find potential occluders. Finally, the fall-off function defines how the distance from the to-be-shaded point to the occluder is used to yield an ambient occlusion value.

Let us proceed by exploring the different real-time ambient occlusion methods that are commonly used in practise.

2.4.1 Baked Ambient Occlusion

The idea behind baked ambient occlusion is to pre-compute an ambient occlusion texture, or to *bake* the ambient occlusion of a scene into a texture, and to later apply the texture at runtime. This method is real-time in the sense that once the ambient occlusion texture is computed, applying it at runtime is as fast and simple as a texture fetch.

Since the ambient occlusion texture is computed offline, methods such as ray tracing are often used, which yield the best results as can be seen in figure 2.11. However, the most immediate downside of this approach is that it can only handle static scenes, since the ambient occlusion computation is done offline. This method cannot therefore represent the ambient occlusion of moving characters or objects, making it rather limited in applications such as games.

In baked ambient occlusion, the 3D representation of the scene is often a ray-scene intersection acceleration data structure, such as a bounding volume hierarchy (BVH) or kd-tree. The sampling scheme is simply a Monte Carlo integration of the ambient occlusion equation using ray tracing, and the fall-off function can be any user-defined function.

FIGURE 2.11: Ray-traced ambient occlusion using Mental Ray. Source: http://artasmedia.com.

2.4.2 Screen Space Ambient Occlusion

Screen space ambient occlusion is a very popular technique, and is the technique that we focus on in this study. As hinted by the name, this method works in screen space and is independent from the number of triangles in the scene. This allows screen space ambient occlusion to be computed in real-time using modern graphics cards and to trivially handle dynamic scenes. As a consequence, screen space ambient occlusion has become an attractive and widely used method in real-time applications such as games. Figure 2.12 shows an example of this method in the game *Assassin's Creed: Black Flag*.

FIGURE 2.12: Screen space ambient occlusion in *Assassin's Creed: Black Flag*. Source: https://developer.nvidia.com.

The main observation behind screen space ambient occlusion is that a pair of depth and normal buffers, or ND-buffer for short, provides a coarse approximation of the 3D geometry of the scene. This concept is illustrated in figure 3.1, where we see the pixels of the depth buffer approximating the scene's geometry, represented with a smooth curve. While this approximation is indeed very coarse, it is enough to provide a relatively good estimate of the ambient occlusion in a scene.

Since screen space ambient occlusion operates on an ND-buffer, the complexity of this method depends solely on the resolution of these buffers and is independent of the number of triangles in the scene. This is in contrast to other ambient occlusion techniques, and is what makes this method so attractive and popular in many domains. In addition, ambient occlusion is relatively low-frequency in most scenes, compared

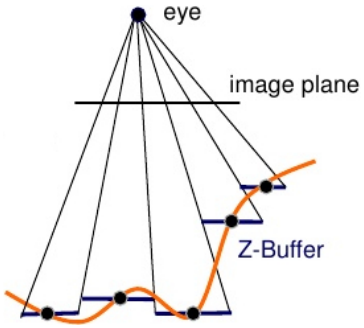
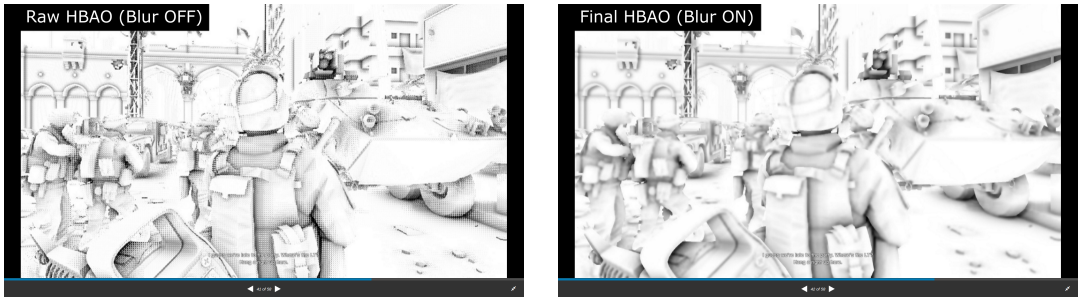


FIGURE 2.13: The depth buffer as an approximation of the 3D geometry of a scene. Source: [Gre09].



(A) Noise produced due to under-sampling. (B) Noise removed using a blur filter.

FIGURE 2.14: Noise produced due to under-sampling in screen space ambient occlusion, then removed using a blur filter. Source: [Gre09].

to other effects such as specular reflections. For this reason, the ND-buffer is often downscaled in practise, making this method even more efficient performance-wise.

The mechanics of screen space ambient occlusion are conceptually very simple. The depth buffer is used to reconstruct the 3D position of a pixel, while the normal buffer gives an approximation of the surface's orientation. The ND-buffer is then stochastically sampled at runtime around the pixel to find nearby occluders, and a fall-off function determines the occlusion value for each sample. Finally, the occlusion values for each sample are all averaged together to produce the final occlusion value for the pixel.

In practise, however, screen space ambient occlusion reveals some complications. For the algorithm to run in real-time, only a few number of samples can be gathered per pixel, leading to under-sampling and producing noisy results like in figure 2.14a. To combat this under-sampling, the resulting ambient occlusion texture is often blurred in a post-process step to remove noise. Figure 2.14b shows the same scene as in figure 2.14a after the blur filter is applied.

Another limitation inherent to screen space ambient occlusion methods is due to the fact that only geometry visible from the camera is considered. Since the input to this set of algorithms is an ND-buffer, geometry that falls out of the ND-buffer is ignored. While this may not produce many visible artifacts for a static image, the artifacts do

show up in the form of flickering or popping pixels when the camera is animated and pieces of geometry pop in and out of the ND-buffer. Some screen space techniques result in less noticeable popping than others, while a few go as far as handling it explicitly in an attempt to improve the visual quality during animation.

2.4.3 Geometry-based Ambient Occlusion

Geometry-based ambient occlusion is the term we use to describe a family of ambient occlusion algorithms that use the 3D geometry of scene to compute ambient occlusion. This family of algorithms can be seen to lie halfway between ray-traced ambient occlusion and screen space ambient occlusion. Unlike screen space techniques, geometry-based methods use geometry to compute ambient occlusion, and therefore do not suffer from sampling artifacts such as under-sampling or popping. However, these methods do not rely on ray tracing, so they are more efficient than ray-traced ambient occlusion and can in fact run in real-time.

The approaches taken by geometry-based methods are varied. In some of these techniques, the scene's geometry is approximated with simple primitives such as spheres, disks, or boxes, as do [SA07] and as illustrated in figure 2.15. In [KL05b], the authors approximate occluders using spherical caps, which are represented with a direction and a solid angle, and then use these to compute ambient occlusion at the occludees. This produces visually plausible results as seen in figure 2.16. In [McG10], the authors extrude the triangles of the scene horizontally and vertically in a geometry shader to produce a volume for each triangle. The volumes are then rasterized, and the fragments generated used to accumulate ambient occlusion values in an output texture.

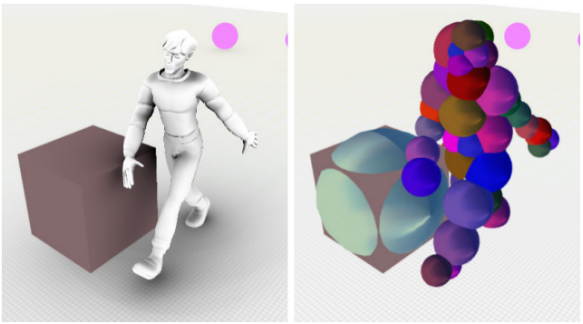


FIGURE 2.15: Approximating the 3D geometry of the scene with spheres.
Source: [SA07].

2.4.4 Volume-based Ambient Occlusion

Volume-based techniques rely on a volume representation of the scene to compute ambient occlusion. The idea behind these approaches is to simplify the 3D geometry of the scene by computing a voxelization and then using the resulting volume to derive

FIGURE 2.16: Ambient occlusion fields. Source: [KL05a].

an AO term for every point to be shaded. The AO term can be computed using ray marching or by sampling a point's nearby voxels, for example.

Volume-based approaches have certain advantages with respect to the other methods. Unlike ray-traced methods, volume-based techniques remain relatively decoupled from the number of triangles in the scene. In practise, however, greater detail will be required in denser areas, so that is why we say *relatively* decoupled. Furthermore, these approaches have a full 3D representation of the scene unlike screen space approaches, so problems such as popping are not fundamental in volume-based approaches.

2.5 Mobile GPU Architecture: Tile-Based Deferred Rendering

Since our work is based on mobile, this chapter would not be complete without an overview of the basics of mobile GPU architecture. This section explores tile-based deferred rendering, the rendering pipeline implemented by mobile chips, and compares it with the traditional immediate mode rendering. Further coverage of this topic can be found in [Som15] and [Mer12].

2.5.1 Immediate Mode Rendering

From a high-level point of view, the graphics pipeline is conceptually very simple. First, the CPU sends a list of triangles to the GPU. Then, the GPU processes these triangles, applying operations on the vertices that define them. Once the triangles have been processed, the GPU rasterises them in order to determine which pixels are covered by each of the triangles. The GPU finally shades each of the pixels to determine their final colour value, and presents the result on screen.

The above process is slightly more complex in reality. Often, a one-to-one mapping between triangles and pixels does not exist. For example, multiple triangles can cover the same pixel, in which case a depth buffer is used to determine their depth order.

Furthermore, if the scene contains transparent surfaces, multiple fragments may contribute to a pixel's final colour by means of alpha blending. Additionally, the GPU often needs to access multiple textures when shading triangles or pixels, such as albedo or normal textures as well as shadow and irradiance maps, thereby adding complexity to the rendering process.

Depth testing, alpha blending, and the myriad of texture fetch operations a GPU must perform per frame result in an increase of memory bandwidth requirements. As discussed in [Mer12], each pixel typically requires a fair amount of bandwidth, often exceeding 100 bytes per pixel. With a resolution of 1920×1080, for example, that would be over $197M$ per frame. Furthermore, many applications have traditionally suffered from *overdraw*, resulting in a further increase of bandwidth requirements. The previous frame, for instance, would consume around $788M$ of memory bandwidth with an overdraw of $4\times$.

While the above rendering scheme may present itself as expensive due to the high bandwidth requirements, this is exactly what immediate mode renderers have been implementing. In immediate mode rendering, the host sends triangles to the GPU and the GPU then processes them, rasterises them, and shades the pixels covered by them in a brute-force manner. As put by [Som15], this is done with no context of what has already happened or what might happen next. As a consequence, immediate mode rendering hardware is relatively simple to design and manufacture, and has ruled desktop GPU architectures for years. Its high-demanding bandwidth has been satisfied with ever-increasing memory speeds and bus widths, allowing immediate mode renderers to scale up.

2.5.2 Tile-Based Deferred Rendering

While immediate mode rendering has been able to scale up on desktop GPUs by improving memory bandwidth, it has however failed to scale down. The high memory bandwidth requirements of immediate mode renderers result in a heavy tax on power consumption and therefore battery life, making them suboptimal on mobile devices. For this reason, a new approach is needed that is lighter on memory bandwidth.

Given that memory is expensive in terms of time, space and power, it makes sense to introduce a memory hierarchy similar to cache memory in CPU architectures to improve on all of these domains. This idea is at the essence of tile-based deferred rendering. In tile-based deferred rendering, the framebuffer, which includes colour, depth, stencil, and multisample buffers, is moved from main memory into a high-speed, on-chip memory. Hardware operations such as vertex shading and hidden surface removal then happen on this on-chip memory, speeding up the rendering process and lowering bandwidth requirements.

Indeed, if the entire framebuffer were able to fit on the on-chip memory, the problem would be solved. This is however not the case. For this reason, tile-based deferred rendering divides the scene into tiles of sizes ranging from 16×16 and 32×32, as illustrated in figure 2.17. A tile-based deferred renderer then processes these tiles one by one, or sometimes in small batches. For each tile, the renderer moves the data needed to render the tile from main memory into the on-chip memory, performs all necessary operations using this high-speed memory, and then moves the final result back to main memory. Since many of the memory read/writes happen on on-chip memory, bandwidth is reduced by only having to access main memory at the beginning and at the end of the process.

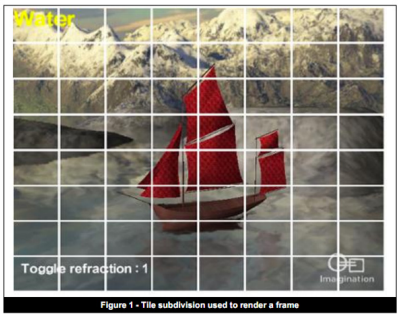


FIGURE 2.17: Screen divided into tiles in tile-based deferred rendering.
Source: [KS11].

Additionally, several stages in the graphics pipeline are deferred in tile-based deferred rendering. For each tile, the hardware performs depth/stencil testing and alpha blending before the triangles are rasterised. Thus, rasterisation is deferred until tiling operations are done. In addition, fragment processing is also deferred, saving the GPU from processing fragments that may later be overwritten. This is what gives tile-based deferred rendering the second part of its name.

Finally, it is worth noting that tile-based deferred rendering is transparent to the programmer. From the programmer's point of view, a tile-based deferred renderer is no different from an immediate mode renderer: the CPU still sends a list of triangles to the GPU, and the GPU still processes these triangles in an apparent brute-force manner.

2.5.3 Performance Guidelines

Even though tile-based deferred rendering is transparent to the programmer, in practise, being familiar with the inner workings of this rendering scheme is essential to optimising mobile applications.

Keeping framebuffer sizes to a minimum is an often followed guideline when developing mobile applications [Ori15]. A tile-based deferred renderer processes tiles one by one or in small batches. For each tile, the renderer moves framebuffer data from

main memory to on-chip memory. As a consequence, the amount of tiles the screen is divided into is a function of the framebuffer size. The smaller the framebuffer, the less tiles needed, and the faster the rendering process is. An application should therefore preferably minimise the size of the framebuffer. If offscreen rendering is used, an application should use as few render targets as possible, and only as much precision as needed for its textures.

As explained in [Mer12], some GPUs such as those by PowerVR may perform hidden surface removal, removing the cost of shading hidden pixels and the associated bandwidth consumption due to memory transfers. In these architectures, performing a Z-prepass does not provide any benefit. However, for most other architectures a Z-prepass may still boost performance, so the programmer should still profile the application and consider doing a Z-prepass like in desktop applications.

Finally, mesh vertex layouts can also be optimised for tile-based deferred rendering. When a tile is processed, triangles are first transformed, depth-sorted, clipped and rasterised before any fragment processing takes place. During vertex processing, no vertex information is required other than the vertex's position; normals, colours, texture indices and other per-vertex information is typically accessed only in fragment shaders. For this reason, a vertex layout that places vertex positions in one buffer and then interleaves the other vertex data in another buffer is optimal for this type of renderers.

Chapter 3

Screen Space Ambient Occlusion

In this chapter, we explore screen space ambient occlusion in greater detail and present the different algorithms our work is based on.

3.1 Screen Space Ambient Occlusion

Screen space ambient occlusion is a screen space technique that approximates the ambient occlusion of a scene. The main observation behind this technique is that a pair of depth and normal buffers, or ND-buffer for short, provides a coarse representation of the geometry of a scene, as illustrated in figure 3.1. While coarse, an ND-buffer provides a good enough representation of the scene that can be used to approximate the scene's ambient occlusion. This is done by sampling a neighbourhood of points around every pixel and using those samples to provide an estimate of the ambient occlusion value for that pixel. In this way, screen space ambient occlusion is a Monte Carlo approximation of the ambient occlusion integral that uses an ND-buffer to determine a pixel's surface position and orientation.

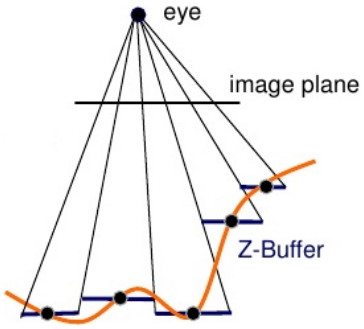


FIGURE 3.1: The depth buffer as an approximation of the 3D geometry of a scene. Source: [Gre09]

In this section we review the different concepts that appear in the discussion of screen space ambient occlusion algorithms.

3.1.1 Near-field and Far-field Ambient Occlusion

One way to categorise ambient occlusion methods is based on their reach or extent. Near-field methods are those that produce a very local effect, in which a point is occluded only by geometry that is close by. On the other hand, far-field methods are those having a far reach and taking a greater neighbourhood into account. This is illustrated in figures 3.2 and 3.3. Figure 3.2 shows a near-field effect. Notice how only the creases in the object are occluded, and how the rest of the geometry appears completely unoccluded. Figure 3.3, on the other hand, shows an example of line-sweep ambient obscurance, a far-field effect described in [Tim13]. The extent of the ambient occlusion is much greater in this method, and even flat walls exhibit some occlusion due to distant geometry.

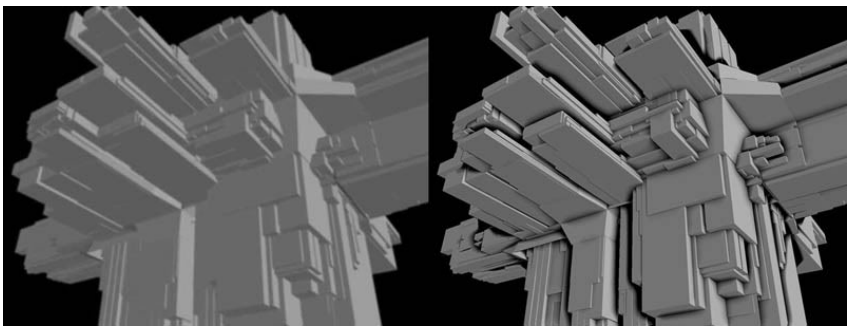


FIGURE 3.2: Unsharp masking the depth buffer produces a near-field ambient occlusion. Image from Mike Pan. http://mikepan.com/

FIGURE 3.3: Line-sweep ambient obscurance is a great example of far-field ambient obscurance. Source: [Tim13]

In section 2.3.4, we saw how the fall-off function ρ assigns smaller values of ambient occlusion to occluders that are farther away and cuts the value to zero when the occludder is at a distance greater than some threshold. The fall-off function then determines how *local* or *global* the ambient occlusion method is, that is, how much of the geometry surrounding a given point influences that point's ambient occlusion value. When only a nearby portion of the geometry influences the point's value, we say that the method computes a near-field ambient occlusion. In contrast, when a far portion of the surrounding geometry is taken into account, we say that the method computes a far-field

ambient occlusion.

In practise, many ambient occlusion methods have their performance limited by the geometrical size of their sampling kernel, that is, how near- or far-field they are. For example, a screen space method will generally have to sample a depth and normal buffer at randomised locations. The wider the sampling kernel is, the more incoherent the texture fetches become, thereby downgrading the algorithm's performance. Volume-based approaches suffer from the same problem. Recent techniques attempt to tackle this problem explicity, such as the line-sweep algorithm presented in [Tim13], but the problem remains for many of the popular methods.

3.1.2 Banding, Noise and Blur

As seen in previous sections, screen space ambient occlusion methods typically can only gather a few number of samples per pixel to be able to run in real-time. This results in two common under-sampling artifacts: banding and noise.

Banding occurs when only a few samples per pixel are gathered and then regions of neighbouring pixels result in similar ambient occlusion values. This produces visible stripes or bands in the output that are visually disturbing, as illustrated on the leftmost image of figure 3.4.

To combat banding, screen space techniques typically randomise the samples taken per pixel. In this way, neighbouring pixels are determined to compute different ambient occlusion values, effectively removing the banding. However, since under-sampling still exists, the combination of under-sampling and randomisation manifests itself as noise in the image, as seen in the middle image of figure 3.4.

To remove the noise resulting from the randomisation of samples, screen space methods often blur their output in a post-process step. The most popular blur method is the *bilateral* filter, which is a Gaussian filter that operates on the resulting ambient occlusion output as well as the depth buffer. The latter is used to avoid blurring the occlusion across the boundaries of objects. Blurring is illustrated in the rightmost image of figure 3.4.

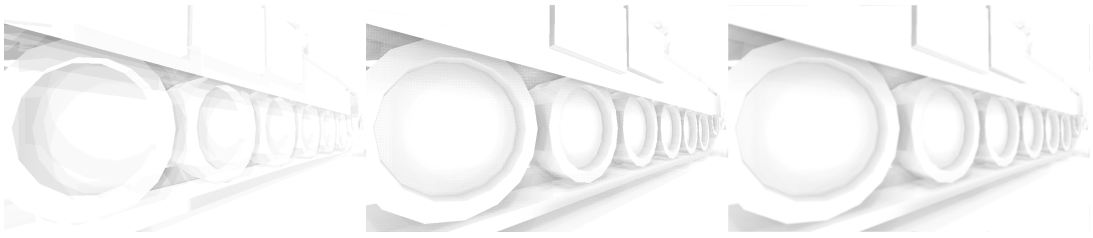


FIGURE 3.4: Banding, noise and blur in our implementation. Random sampling fixes banding artifacts but introduces noise. The noise is blurred away in a post-process step, resulting in the rightmost image.

3.1.3 Popping

Banding and noise are not the only artifacts that result from screen space ambient occlusion methods. *Popping* is also a common artifact. Since these methods work in screen space, geometry that is not visible to the camera does not cast occlusion on nearby objects. However, as the camera moves, these objects may suddenly become visible and cast occlusion on nearby surfaces, in which case shadows appear to pop into the image. Similarly, when this geometry goes back out of scope, the shadows that they cast pop back out. Popping is visually disturbing and cannot unfortunately be reproduced on print, since a camera must be animated for popping to occur.

3.1.4 Flickering

Flickering is another artifact resulting from screen space approaches. Since screen space ambient occlusion methods sample a number of pixels around the point being shaded, when the camera moves, the set of pixels that are sampled refer to different 3D points in the scene. Depending on the algorithm, this may give the point being shaded a slightly different occlusion value. As a consequence, pixels may appear to flicker during animation, an effect that is especially visually disturbing for the human eye.

3.1.5 Robustness

Ambient occlusion is technically view-independent. However, screen space ambient occlusion is inherently view-dependent, resulting in artifacts such as banding, noise, popping or flickering. A screen space ambient occlusion algorithm is commonly said to be *robust* when it minimises this view dependence. As a consequence, algorithms that are robust are the most visually appealing to the viewer.

To alleviate popping and flickering, an often used technique is the use of guard bands. Guard bands increase the size of the offscreen viewport where the ambient occlusion computation is performed so that more geometry than the camera can see is taken into account. With guard bands, if the original viewport has a size of $N \times M$, the ambient occlusion viewport would have a size of $(N + G) \times (M + G)$, where G is the size of the guard bands. In this way, more geometry is effectively taken into account, and both popping and flickering are potentially reduced.

3.1.6 Scalability

Screen space ambient occlusion techniques often take many parameters, such as the sample kernel radius, the number of samples, or resolution. Often, screen space techniques are said to be *scalable* when these parameters can be improved without a significant penalty in performance. Some methods scale down — can run in less capable hardware with the same parameters and similar performance — while others scale up — can increase the sample radius, number of samples or resolution without considerably hurting performance.

3.2 Screen Space Ambient Occlusion Techniques

In this section we present the different screen space ambient occlusion techniques our work is based on.

3.2.1 Image Enhancement by Unsharp Masking the Depth Buffer

Although not a true ambient occlusion technique, and although not physically-based, the method presented in [LCD06] results in an image enhancement effect similar to ambient occlusion. The essence of this technique is to map depth discontinuities in the depth buffer to a scalar so as to darken crevices in the scene's geometry. An example of this technique can be seen in figure 4.21.

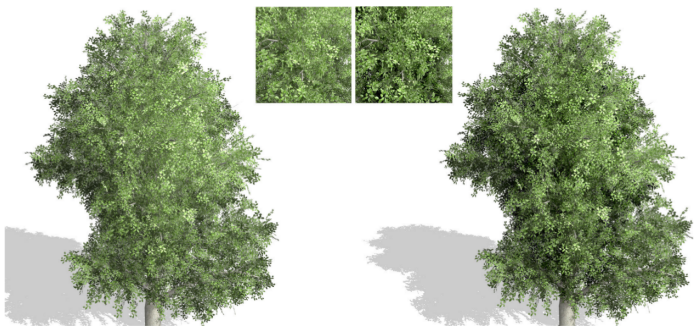


FIGURE 3.5: Unsharp masking of the depth buffer. Source: [LCD06].

Although the paper presents several techniques to modulate the colour, luminance or contrast of an image, the basic pipeline remains the same for all techniques. First, the algorithm blurs the depth buffer by applying a Gaussian filter, and then subtracts the result from the original depth buffer:

$$\Delta D = G * D - D$$

where D is the depth buffer and $G * D$ is a the convolution of a Gaussian filter kernel and the depth buffer D. By subtracting the blurred depth buffer from the original, the algorithm produces a new buffer ΔD that contains the high frequencies of the depth buffer D. In other words, if a discontinuity existed in the original depth buffer D, the discontinuity is enhanced in ΔD.

The algorithm then proceeds by using ΔD to alter the original colour image I in some way. In its basic form, the unsharp mask method simply subtracts a scaling of the resulting ΔD texture from the colour image I to achieve its darkening effect:

$$I' = I + \lambda \Delta D$$

for some $\lambda \in \mathbb{R}$.

As mentioned above, unsharp masking of the depth buffer is not technically an ambient occlusion method, but it produces results that can be understood as an approximation to true ambient occlusion. The main advantage of this technique with respect to true ambient occlusion methods is that it remains simple and computationally inexpensive, making it suitable for lower end devices and especially suitable for mobile devices.

3.2.2 Crytek Ambient Occlusion

Screen space ambient occlusion was originally developed by Vladimir Kajalin at Crytek. The technique was integrated into *CryEngine 2*, the game engine powering the popular 3D game *Crysis*, which landed on the market in 2007. The technique was then published as a SIGGRAPH course in that same year under the name *Finding next gen: CryEngine 2* [Mit07]. Being the first screen space technique, Crytek ambient occlusion remains one of the most popular methods, since it was the pre-cursor to the myriad of techniques that exist nowadays. A screenshot showing the result of Crytek ambient occlusion can be seen in figure 3.6.

FIGURE 3.6: Screen space ambient occlusion in Crysis. Source: Wikipedia.

Perhaps the most notable characteristic of Crytek ambient occlusion is the resulting greyish images that it produces, as shown in the previous figure. This is a side effect of how the method determines a point's ambient occlusion value. In this method, potential occluders centred in a sphere around a given point are sampled to determine the point's occlusion value. The samples are then projected to texture space, and their depth compared to the depth value stored in the position of the depth buffer where they project to. The two depth values are compared to determine if the sample is occluded by geometry (the sample's depth is greater than the depth value stored in the depth buffer), and the occlusion factor is finally defined as the proportion of samples that are indeed occluded. This process is illustrated in figure 3.7. Because the sampling kernel is spherical, half of the points are deemed to lie behind the surface for planar surfaces, resulting in an over-occlusion value of $\frac{1}{2}$ for these surfaces and yielding an overall gray image.

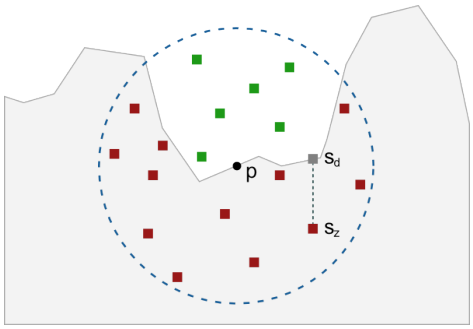


FIGURE 3.7: Sampling kernel in Crysis ambient occlusion. Samples are taken within a hemisphere centred at the point being shaded. As a consequence, flat surfaces appear gray, since half of the samples are occluded for such surfaces. Source: [Aal13].

From the algorithm description above, we can conclude that Crytek ambient occlusion is a depth-only technique. That is, the only input to the Crytek method is a depth buffer; no normal buffer is used. This saves bandwidth both when creating the off-screen buffers and when sampling a point's neighbours in the ambient occlusion shader pass, but makes the method unable to determine surface orientation. Even though normals could be reconstructed from the depth buffer by computing derivatives, this would induce an additional overhead in the ambient occlusion pass in terms of computation and additional texture fetch instructions, so the original algorithm does not do this. The lack of surface orientation results in a spherical kernel, which in turn gives this method its characteristic greyish look.

While the method's output could be considered aesthetically pleasing, the worst part comes when analysing its performance. Since half of the samples are deemed to lie behind the surface for planar surfaces, the algorithm essentially wastes half of its samples. Modern techniques simply fix this issue by using a normal buffer to determine surface orientation, the cost of which is no longer prohibitive in modern GPUs.

Finally, Crytek ambient occlusion uses a blur pass to reduce noise in its output. Crytek ambient occlusion gathers 12 to 32 samples per pixel, depending on the machine's performance, which would in any case result in an under-sampling of the ambient occlusion function that would in turn manifest itself as banding artifacts. To combat this issue, the algorithm randomises the samples taken per pixel, trading banding for noise. This noise is then removed in a secondary pass using a bilateral filter, resulting in smooth images such as the one in figure 3.6. This additional blur pass has remained quintessential of screen space ambient occlusion methods to reduce the noise resulting from under-sampling.

3.2.3 Image-Space Horizon-Based Ambient Occlusion

Horizon-based ambient occlusion (HBAO) is a technique developed by Louis Bavoil and Miguel Sainz at NVIDIA and introduced at SIGGRAPH in 2008 [BSD08]. The observation behind this approach is that if we trace a horizon line from a point p in some direction θ, then rays below the horizon are known to be occluded under the assumption that the heightfield is continuous, in which case their ambient occlusion evaluation can be skipped. This is illustrated in figure 3.8.

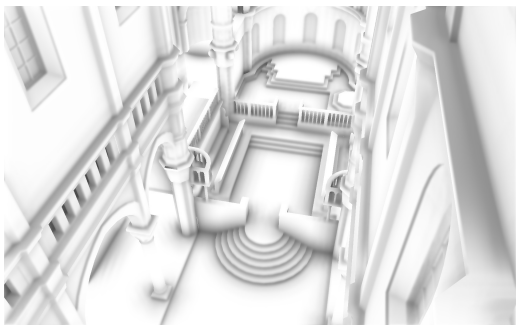


FIGURE 3.8: Horizon-based ambient occlusion. Source: [BSD08].

Note that the horizon angle depends on the direction θ in which the ray is traced from the point p. For this reason, the horizon angle must be computed along all directions θ in the normal-oriented hemisphere. The ambient occlusion equation for horizon-based ambient occlusion therefore becomes a double integral:

$$AO(p) = 1 - \frac{1}{2\pi} \int_{\theta=-\pi}^{\pi} \int_{\alpha=t(\theta)}^{h(\theta)} W(\omega) \cos\alpha \, \mathrm{d}\alpha \, \mathrm{d}\theta$$

In its implementation, horizon-based ambient occlusion is computed by ray marching the depth buffer. A set of rays are shot from the point being shaded in randomised directions. For each ray, the algorithm marches the depth buffer along the ray's direction and reads from the depth buffer at discretised positions. Every time the algorithm finds a point that is closer to the camera than the currently known closest point, the

point is treated as an occluder and the ambient occlusion function is evaluated for that point. On the other hand, points that do not satisfy this criteria can be skipped, since they are known to have been occluded by the currently known closest point. Finally the individual ambient occlusion terms for each sample are all added together to yield the point's final occlusion value.

As is usual in ambient occlusion algorithms, samples for each pixel are randomised to prevent banding artifacts. In horizon-based ambient occlusion, this translates to randomising the ray directions for every pixel. However, this randomisation and the relatively low number of samples results in an under-sampling of the ambient occlusion function, yielding to variance that manifests itself as noise in the image. To combat this variance, horizon-based ambient occlusion performs a blur in a post-process step to remove the noise.

An example of horizon-based ambient occlusion can be seen in figure 3.9 applied to the famous dragon model.

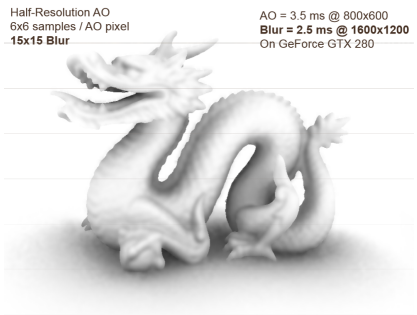


FIGURE 3.9: Horizon-based ambient occlusion on the dragon model. Source: [BSD08].

3.2.4 Starcraft 2 Ambient Occlusion

2008 also saw the release of *Starcraft 2: Effects and Techniques*, an article published in SIGGRAPH describing the ambient occlusion technique featured in the game *Starcraft 2* [FM08]. Starcraft 2 ambient occlusion can be seen as an improvement over Crytek ambient occlusion, in which the surface normal is used in determining the surface's orientation to avoid sampling points behind the surface. This yields images like the one in figure 3.10. Notice that in Starcraft 2 ambient occlusion, flat surfaces such as the floor or bar appear white, unlike in Crytek ambient occlusion, in which they would appear gray.

Starcraft 2 ambient occlusion introduces two contributions with respect to Crytek's method. The first improvement is that the algorithm uses the surface normal, stored in a normal buffer in a G-buffer pass, to determine the surface's orientation. In this technique, points are sampled in the normal-oriented hemisphere instead of a sphere,

FIGURE 3.10: Screen space ambient occlusion in Starcraft 2. Source: [FM08].

as shown in figure 3.11. This has two implications. First, flat surfaces appear white, since points are now sampled only in front of the surface. This is closer to what a ray-traced ambient occlusion method would produce and is more visually appealing. Second, all of the samples gathered in Starcraft 2 are actually meaningful, since they are known to lie in front of the surface. In other words, Starcraft 2 does not apply wasted effort sampling points behind a surface.

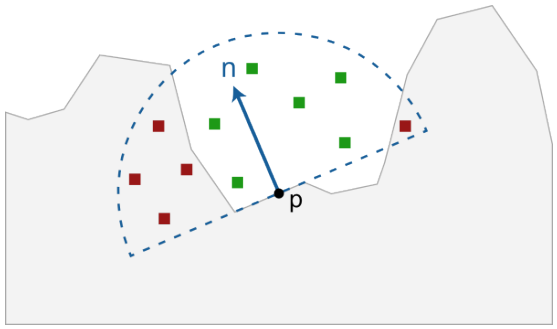


FIGURE 3.11: Sampling kernel in Starcraft 2 ambient occlusion. Samples are taken within a normal-oriented hemisphere centred at the point being shaded. Unlike Crysis ambient occlusion, flat surfaces are now shaded correctly and appear white. Source: [Aal13].

The second contribution from Starcraft 2 ambient occlusion is its special fall-off function. In fact, the article proposes not a function in particular, but a family of functions. One such function is depicted in figure 3.12. The idea behind this family of functions is that the ambient occlusion for a point should be a function of the depth difference between a sample's depth value and the value stored in the depth buffer at the sample's projection. This is in contrast to Crytek's ambient occlusion method, where the test is simply boolean.

To be physically-correct, the occlusion should fall as an inverse square of the depth difference between a sample's depth and the depth stored at the sample's projection at the depth buffer. However, the authors choose to give their artists more freedom by allowing them to choose any power curve as long as it satisfies three criteria. First, the curve should be 0 for negative depth deltas, that is, for samples that are determined to be

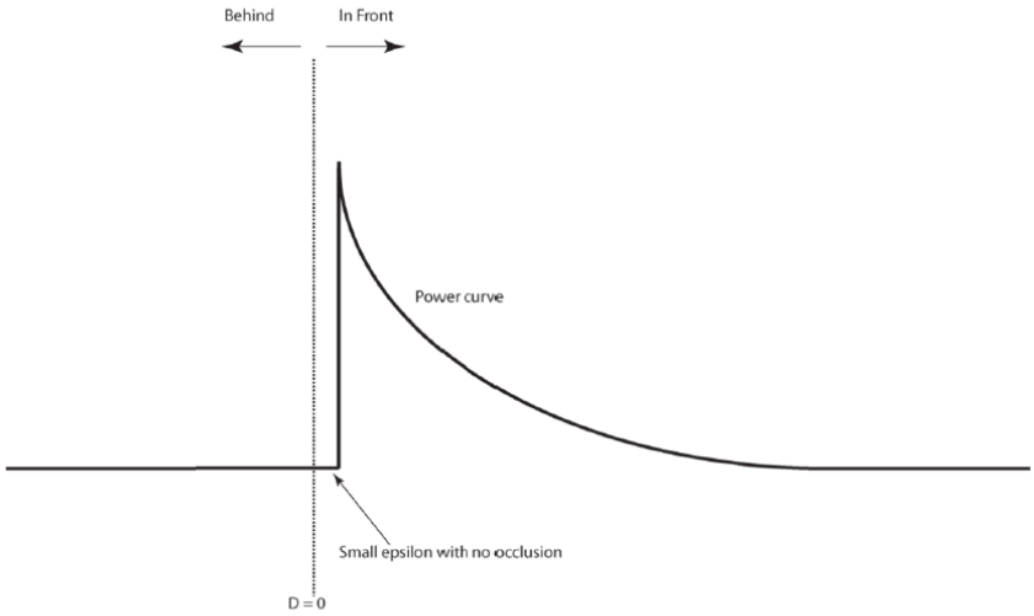


FIGURE 3.12: Falloff function used in Starcraft 2 ambient occlusion. Source: [FM08].

unoccluded. Second, the curve should yield greater occlusion values for smaller depth deltas, mimicking the fact that nearby occluders produce greater occlusions. Finally, the curve should fall off to 0 for large deltas for the same reason.

In addition to the above three criteria, the authors propose a fourth addition to the fall-off function. The authors use a small epsilon to prevent false occlusions. Since the depth and normal buffers have only a limited precision, in practise, samples outside of the normal-oriented hemisphere and behind the surface may be taken. To prevent these samples from contributing to the ambient occlusion of a point, the authors design their fall-off function such that small depth deltas below the threshold yield an occlusion of zero, instead of yielding maximum occlusion. This prevents false occlusions, making flat surfaces appear completely unoccluded as expected.

Finally, it is worth noting that the idea of using a normal buffer to determine surface orientation has been adopted by many of the screen space ambient occlusion techniques following Starcraft 2 AO, as well as the use of the family of fall-off function described above. For these reasons, Starcraft 2 ambient occlusion remains one of the most popular techniques.

3.2.5 Screen Space Ambient Occlusion using Temporal Coherence

Screen space ambient occlusion using temporal coherence, or TSSAO for short, is an ambient occlusion algorithm that exploits temporal coherence to reduce noise and blurring artifacts, as well as to speed up previous algorithms [MSW10]. In this approach, ambient occlusion information from previous frames is reused in consecutive frames

by exploiting temporal coherence. Pixels describing identical world-space positions are identified by means of temporal reprojection. The current state of the solution is cached in what the authors call an *ambient occlusion buffer*. In a given frame, ambient occlusion samples are computed, and the results blended with the buffer to yield a final ambient occlusion value for each pixel. In this way, information from previous frames is used in consecutive frames, resulting in a robust algorithm.

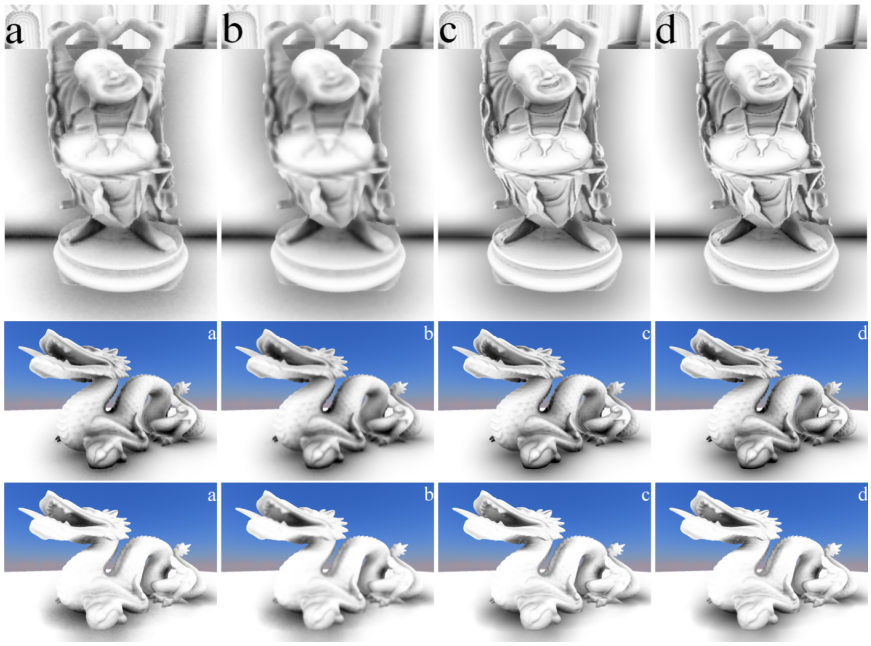


FIGURE 3.13: Temporal screen space ambient occlusion. Source: [MSW10].

3.2.6 Alchemy Ambient Obscurance

The Alchemy ambient obscurance algorithm is a popular technique originally developed for a *Guitar Hero* title and then integrated into Alchemy engine from Vicarious Visions in 2011 [McG+11]. Alchemy chooses a special fall-off function that cancels several terms in the visibility integral and results in an algorithm that is robust, temporally coherent and efficient. An example of Alchemy ambient occlusion can be seen in figure 3.14.

From the author's point of view, Alchemy satisfies four different properties that none of the previous ambient occlusion methods satisfied simultaneously; Alchemy is robust, multiscale, scalable and provides artist control. Robustness refers to the fact that Alchemy produces no halos or false shadows near silhouettes, limits viewer dependency (ambient occlusion is technically viewer independent, but since these methods are screen space they do result in viewer-dependent artifacts) and maximises temporal coherence. Alchemy is also multiscale, meaning that it captures both low-frequency and high-frequency occlusion details. In addition, Alchemy's fall-off function allows

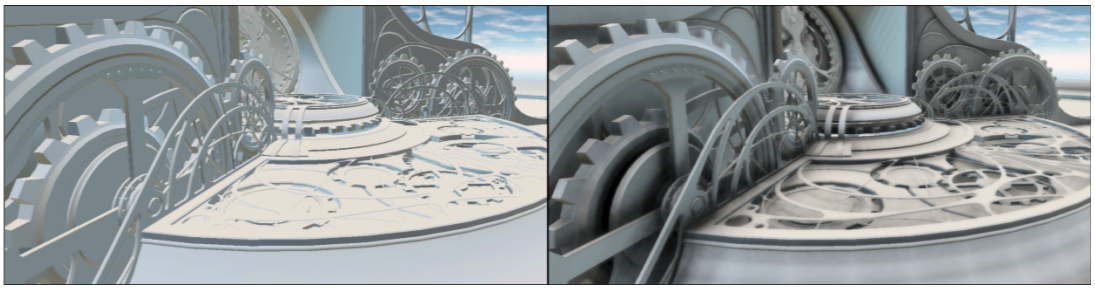


FIGURE 3.14: Alchemy ambient occlusion in an example scene. Source: [McG+11].

the artist to control four different parameters that are relatively independent from each other: radius, bias, intensity and contrast. Finally, Alchemy is scalable, computed in 3 to 5ms in a range of devices from the Xbox 360 to Direct3D 11 hardware.

All of the above properties are a consequence of Alchemy's ambient occlusion equation. After making all of the simplifications and applying Monte Carlo on the visibility integral, Alchemy's equation becomes

$$AO(p) = max\left(0, 1 - \frac{2\sigma}{s}\sum_{i=1}^{s}\frac{max(0, \vec{v}_i \cdot \vec{n} + z_C\beta)}{\vec{v}_i \cdot \vec{v}_i + \epsilon}\right)^k$$

where s is the number of samples, $\vec{v}_i$ is the vector from the occluder to the occludee, σ is the intensity scale, k is the contrast, $z_C\beta$ is the product of the usual depth bias β and the occludee's depth value z_C, and ϵ is a small number to prevent underflow.

The sampling scheme in Alchemy ambient occlusion is simple and straightforward. Samples are generated in texture space in a disk centered at the projected of the to-be-shaded point p. These samples are then unprojected to view space, yielding potential occluders around the point p. The term $\vec{v}_i$ in the equation is then defined as the vector from p to the potential occluder, the equation is applied, and the results averaged for every sample. This procedure is illustrated in figure 3.15.

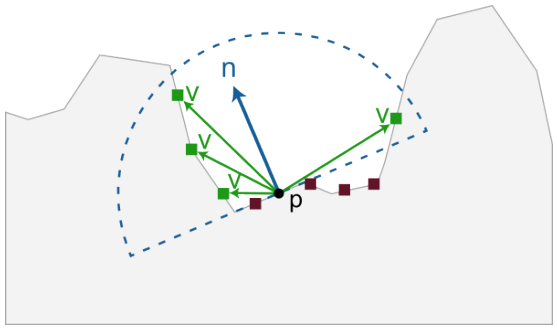


FIGURE 3.15: Sampling kernel used in Alchemy ambient obscurance. Samples are taken from a normal-oriented hemisphere centred at the point being shaded. Source: [Aal13].

3.2.7 Separable Approximation of Ambient Occlusion

In *Separable Approximation of Ambient Occlusion* [Hua+11], the authors note that screen space ambient occlusion can be understood as a local 2D filter that evaluates a point's visibility in screen space. Inspired by the popular separable Gaussian filter in the image processing domain, the idea behind separable ambient occlusion is to separate this 2D filter into two 1D filters. While this approach is not technically correct and only results in an approximation of the original 2D filter, the method is an order of magnitude faster and the results remain visually appealing, as illustrated in figure 3.16.

FIGURE 3.16: Examples of separable ambient occlusion. Source: [Hua+11].

Separable ambient occlusion can be combined with previous ambient occlusion methods. This is because separable ambient occlusion only modifies the sampling scheme of an ambient occlusion technique, and does not impose any particular ambient occlusion equation. This gives the developer the freedom to choose their equation, and as a result, separable ambient occlusion may result in many different implementations in practise.

In separable ambient occlusion, samples are gathered by ray marching the depth buffer along two orthogonal line segments anchored at the point to be shaded. An ambient occlusion equation is applied to these samples, and the results averaged together to produce the point's final occlusion value. Because scanning along to fixed directions for every pixel would produce banding artifacts, in practise, the set of orthogonal directions is randomised for every pixel with the help of a texture of random orthogonal vectors, as illustrated in figure 3.17. As with other techniques, this randomisation trades banding for noise, in which case the results must be blurred in a post-process step.

3.3 Blur Techniques

Many ambient occlusion techniques rely on a blur filter to remove noise introduced due to random sampling. In this section, we cover two of the most popular blur methods: the bilateral filter and the separable blur.

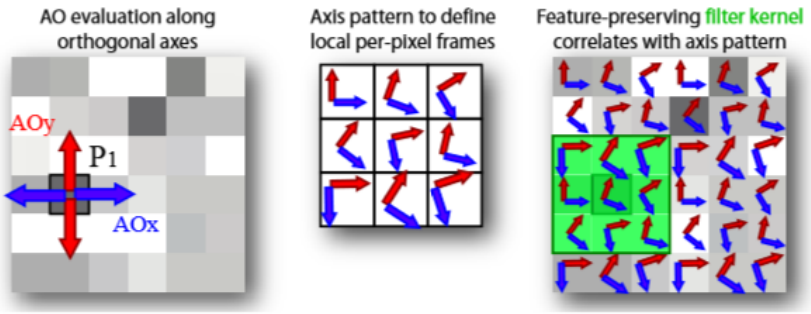


FIGURE 3.17: Sampling patterns in separable ambient occlusion. Ambient occlusion is evaluated for every pixel along orthogonal axes. To improve quality, the orthogonal axes are randomly rotated per pixel. Source: [Hua+11].

3.3.1 Bilateral Filter

The bilateral filter is an extension of the well-known Gaussian blur. A Gaussian blur produces smooth results by blurring each pixel as a function of both the value of neighbouring pixels and their distance to the pixel being blurred. That is:

$$GB[I]_p = \sum_{q \in S} G_\sigma(||p - q||) I_q \tag{3.1}$$

where p is the pixel being blurred, S is a circular neighbourhood of pixels around p, q is a pixel in that neighbourhood, $||p - q||$ is the distance from p to q, I_p is the intensity of pixel p, I_q is the intensity of pixel q, and G_σ is the normalised Gaussian function:

$$G_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

where σ is the window size.

Let us illustrate the above definition of the Gaussian blur. Figure 3.18 shows an example Gaussian kernel. Here, a circular neighbourhood S of pixels is shown. The pixel being blurred, p, would be the center of the circle, and q would be each of the pixels in the circle. In addition, every pixel is coloured in a shade of gray. White denotes that the pixel has maximum contribution to the blur, whereas black denotes no contribution. Note that the contribution neighbours is maximum and the center and quickly decreases as we move away from it.

On the other hand, figure 3.19 shows the Gaussian function G_σ in equation 3.1. This is the function that gives pixels their colour in figure 3.18. Note how the function is maximum at the center (where $q = p$) and quickly decreases as we move away from it.

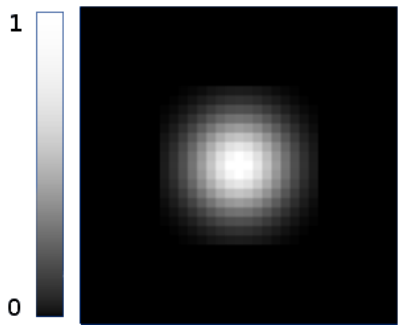


FIGURE 3.18: An example Gaussian kernel. Note how the contribution of neighbours is maximum at the center and quickly decreases as we move away from it. Source: [SPD07]

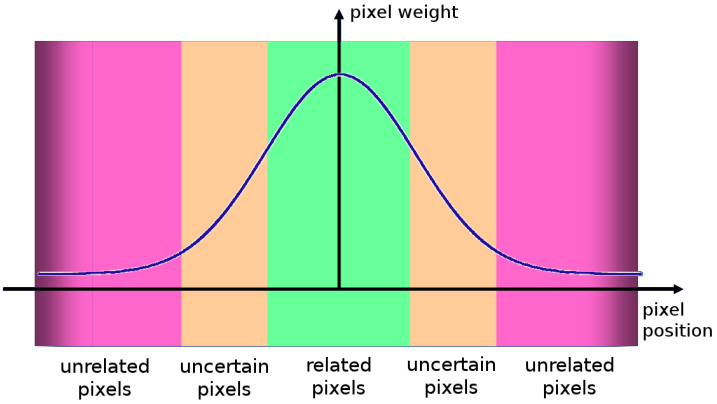


FIGURE 3.19: An example Gaussian function. Note how the contribution of neighbours is maximum at the center and quickly decreases as we move away from it. Source: [SPD07]

Finally, figure 3.20 shows a Gaussian blur applied to an example image. Note how the result is smooth, with no visible artifacts.

In the domain of computer graphics, the Gaussian blur often comes rather close but does not produce the desired result. The problem with the Gaussian blur is that it does not distinguish between pixels of different objects in the scene. In other words, the Gaussian blur is not geometry-aware. As a consequence, the edges of objects are not respected and are blurred away, as illustrated in the right image of figure 3.20.

The solution to the above problem is to blur a pixel by taking into account not just the distance to its neighbours, but also the difference between pixel intensities. This gives rise to the bilateral filter equation:

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(||p - q||)\, G_{\sigma_r}(|I_q - I_p|)\, I_q \tag{3.2}$$

Equation 3.2 is like equation 3.1 but with two additional terms. The term $G_{\sigma_r}(|I_q - I_p|)$

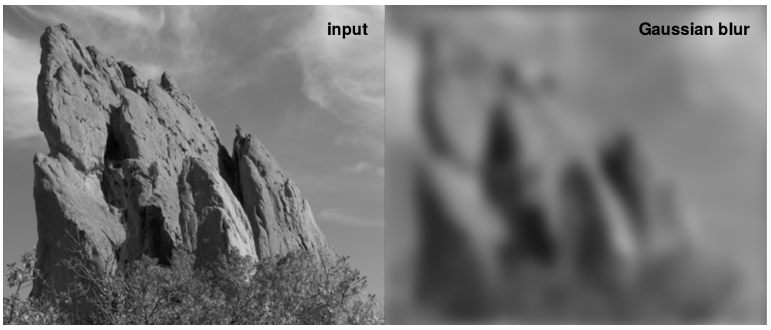


FIGURE 3.20: An example of Gaussian blur applied to an image. Left: input, right: output. Source: [SPD07]

is the same Gaussian function G applied to the difference between the intensities of the neighbour q and the pixel being blurred p. This is what prevents blurring across edges. Furthermore, the term $\frac{1}{W_p}$ is introduced as a normalisation factor that makes the weights add up to 1. Finally, note that we now define two window sizes, σ_s and σ_r, for the space and intensity weights respectively.

Figure 3.21 shows a bilateral filter applied to the same image in figure 3.20. Notice how edges are now perfectly preserved, while the pixels within the contours of objects are still correctly blurred.

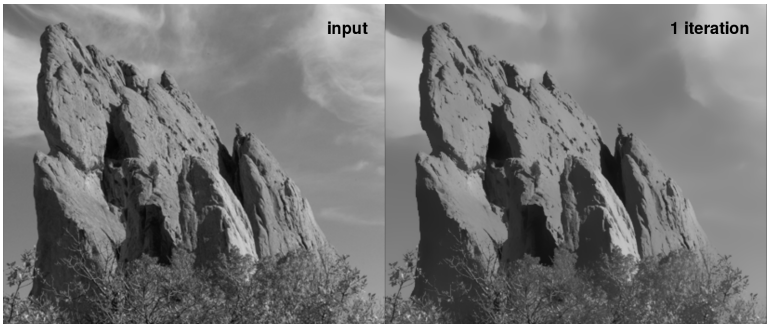


FIGURE 3.21: An example of Gaussian blur applied to an image. Left: input, right: output. Source: [SPD07]

In the context of ambient occlusion, we wish to blur the output of an ambient occlusion shader to remove noise due to random sampling. In this context, the second weight term is not a function of pixel intensity, but a function of pixel depth:

$$BF_{AO}[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(||p - q||)\, G_{\sigma_r}(|D_q - D_p|)\, I_q \tag{3.3}$$

In equation 3.3, D_q is the depth of pixel q and D_p is the depth of pixel p. The intuition behind this equation is that pixels belonging to different objects are often far apart in the plane or far apart in depth, in which case one of the two weight terms in equation 3.3 becomes very small and prevents the bilateral filter from blurring across edges.

Finally, figure 3.22 shows a bilateral filter applied to the output of an ambient occlusion shader to remove the noise due to random sampling. Notice how the noisy patterns are removed, while edges are preserved.

FIGURE 3.22: A bilateral filter applied to the output of an ambient occlusion shader to remove the noise due to random sampling.

3.3.2 Separable Blur

While the bilateral filter provides excellent results, it does have a major drawback: performance. The bilateral filter must access a number of pixels that is proportional to the square of the radius of the filter's kernel, making the algorithm $\mathcal{O}(N^2)$. In the domain of screen space ambient occlusion, this cost is often too high on low-end devices.

The idea behind the separable bilateral filter, or separable blur for short, is to separate the original algorithm into multiple passes, one for each dimension [PV05]. In the context of blurring an image, a first pass blurs the original image along the x-axis to generate an intermediate image, and a second pass blurs this intermediate image along the y-axis.

In mathematical terms, the image is first blurred along the x-axis as if a bilateral filter were being performed, except that the neighbourhood is now defined as a flat stripe of pixels instead of a circle:

$$I'_p = \frac{1}{W_p} \sum_{q \in X} G_{\sigma_s}(||p - q||)\, G_{\sigma_r}(|D_q - D_p|)\, I_q$$

where X denotes a a horizontal stripe of pixels centred at p.

The resulting intermediate image I' is then blurred along the y-axis in a similar manner:

$$I'' = \frac{1}{W_p} \sum_{q \in Y} G_{\sigma_s}(||p - q||)\, G_{\sigma_r}(|D_q - D_p|)\, I'_q$$

Putting both equations together yields the separable blur equation:

$$I'' = \frac{1}{W_p^2} \sum_{q \in Y} G_{\sigma_s}(||p-q||)\, G_{\sigma_r}(|D_q - D_p|) \left(\sum_{q \in X} G_{\sigma_s}(||p-q||)\, G_{\sigma_r}(|D_q - D_p|)\, I_q \right)$$

The separable blur blurs the image horizontally in one pass, and vertically in a second pass. The resulting algorithm is therefore $\mathcal{O}(N)$, where N is the size of the neighbourhood or stripe.

Note that technically speaking, the bilateral filter is non-separable. For this reason, the separable blur only provides an approximation of a true bilateral filter. Nevertheless, in the context of screen space ambient occlusion, this approximation is often good enough, and is cheaper to compute than a true bilateral filter.

Chapter 4

Implementation on Mobile

In this chapter, we describe the different implementations of screen space ambient occlusion that we have written for mobile, the rendering pipeline that supports them and the sampling schemes and blur filters used. We also state the design decisions behind our work and the characteristics and limitations of mobile GPUs on which they are based.

This chapter presents the end result of our work and has been conveniently written in a style that makes it suitable to be read as a reference. In the following chapter, we show the experiments performed during the development of our work to justify the design decisions stated here.

4.1 Characteristics and Limitations of Mobile GPUs

Mobile GPUs exhibit some limitations with respect to desktop GPUs that directly affect the implementation of screen space ambient occlusion. These can be summarised as follows:

- Limited memory bandwidth.
- Limited compute power.
- Very high screen resolutions.

It is worth noting that while these limitations may also exist on desktop GPUs, their effects on mobile GPUs are typically exaggerated.

In our work, we design a rendering pipeline and sampling schemes to help overcome these limitations. To add complexity, optimisations for each of these limitations often conflict with each other. As a consequence, the target application must be profiled and a balance between these conflicting optimisations must be found to achieve optimal performance.

4.1.1 Limited Memory Bandwidth

Unlike desktop GPUs, mobile GPUs must be carefully optimised not only for performance, but also for power consumption, as this is what determines a device's battery life. As it turns out, one of the main factors driving power consumption is memory bandwidth [Amd; Fol14]. As a consequence, mobile devices typically offer very limited memory bandwidth compared to desktop GPUs.

When implementing screen space ambient occlusion, we strive to keep memory bandwidth use to a minimum for performance. This can be achieved by following the guidelines below:

- Minimise texture resolutions and compress textures if possible.
- Use non-native screen resolutions.
- Minimise render target precision.
- Compress G-buffer data.
- Limit the number of samples taken when computing ambient occlusion.
- Prefer linear over quadratic blurs.

The above guidelines are taken from resources such as [Ios] and [QT15]. While these resources are written for specific platforms, the optimisations described often apply to other devices.

4.1.2 Limited Compute Power

As will be seen in the following chapter, not all memory bandwidth optimisations result in a faster rendering pipeline. Some memory optimisations require that additional floating-point operations be performed in shaders, and these may result to be more taxing than the additional use of memory bandwidth. Compared to desktop GPUs, mobile GPUs only offer limited compute power. For example, previous generation NVIDIA GTX 760 offers 2.258 TFLOPS of computing power, whereas the recent Adreno 430 GPU (2015) offers 0.38 TFLOPS[1]. A balance between memory savings and compute power requirements must therefore be established for optimality.

The following guidelines help overcome the limited compute power offered by mobile devices in our implementation:

- Cache data in the G-buffer to avoid additional computations in shaders.
- Limit the number of samples taken when computing ambient occlusion.
- Prefer linear over quadratic blurs.

[1] https://versus.com/en/qualcomm-adreno-430-vs-nvidia-geforce-gtx-760

Note that the first guideline, caching data in the G-buffer, conflicts with the memory-saving guideline of compressing G-buffer data in the previous section. For example, one might decide to compress normals into just two texture channels of a render target. In doing so, however, any shader using normal data must then reconstruct the missing coordinate, increasing the number of floating-point operations performed and drawing more computational power. Ultimately, the target application must be profiled to determine which approach is best.

On the other hand, guidelines such as taking fewer samples during the ambient occlusion computation are shared by both optimisation domains. Taking fewer samples reduces the number of texture fetch operations as well as the number of floating-point operations performed, simultaneously optimising for both memory bandwidth and compute power.

4.1.3 High Screen Resolutions

Mobile devices often pack high screen resolutions into small form factors. For example, Google's Nexus 10 offers a resolution of 2560×1600 pixels in a 10" screen[2], resulting in a higher PPI than that of the average desktop monitor. Such high resolutions directly impact the performance of different stages of the rendering pipeline, increasing memory bandwidth requirements, compute power requirements and adding tax on fillrates.

To overcome this problem in our implementation, we render the scene using non-native resolutions. To avoid deformation, we use a scaling of the screen's native resolution, typically $\frac{1}{4}$ of the original size.

With these limitations in mind, we now proceed to describe our implementation in the sections that follow. Results, performance analyses and justifications for the decisions taken here are provided in the following chapter.

4.1.4 Tile-Based Deferred Rendering

As described in chapter 2, mobile GPUs implement a tile-based deferred rendering pipeline. One of the consequences of this approach is that framebuffer sizes have a direct impact on performance [Ori15]. Larger sizes imply a greater number of tiles, and since these are processed sequentially on in small batches, this implies longer rendering times.

In the context of screen space ambient occlusion and offscreen rendering in general, smaller G-buffers and non-native resolutions are preferred. The former reduces the amount of data needed to process a pixel, and the latter reduces the overall amount of work to be performed on the hardware.

[2] https://en.wikipedia.org/wiki/Nexus_10

4.2 Rendering Pipeline

Before delving into the implementation of each ambient occlusion algorithm, we describe the rendering pipeline supporting these algorithms. We propose two rendering pipelines, each with their own share of advantages and disadvantages. One should choose one of the two pipelines based on the specifics of their application and target platforms, balancing the benefits and downsides of each.

4.2.1 ND-buffer Pipeline

The first pipeline we propose is illustrated in figure 4.1. In this pipeline, the scene's geometry is rendered in a first pass to generate an ND-buffer. An ambient occlusion shader is then invoked, taking the ND-buffer as input and producing an ambient occlusion texture in its output. Finally, the scene's geometry is rendered once again to compute all of the scene's illumination in a forward pass, using the the generated ambient occlusion texture to modulate ambient light.

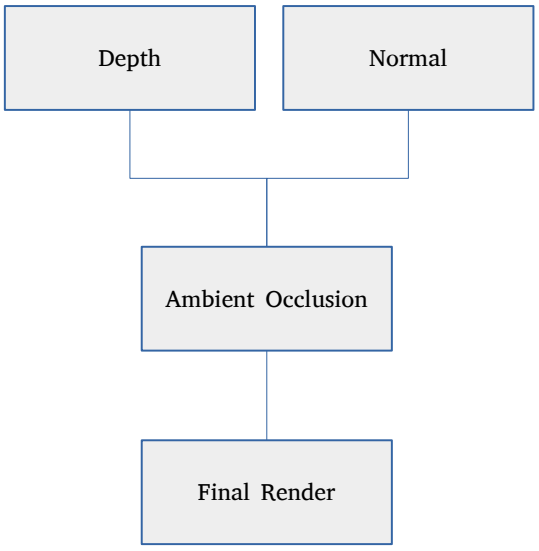


FIGURE 4.1: Ambient occlusion ND-buffer rendering pipeline. The scene is rendered once to generate an ND-buffer. An ambient occlusion is invoked in a second stage to generate an ambient occlusion texture from this buffer. The scene is then rendered in a second and final forward pass where the scene's illumination is computed and the previously generated ambient occlusion texture is used to modulate ambient light.

To generate the ND-buffer, we render the scene's geometry into a framebuffer object. The framebuffer object contains two render targets: a single RGB colour target (`GL_COLOR_ATTACHMENT0`) to store normals and a depth target (`GL_DEPTH_COMPONENT`)

to store the depth. Because these two targets must be sampled from the ambient occlusion shader, we attach textures and not renderbuffers them.

From our experiments, generating an ND-buffer is faster than generating a full G-buffer. Since only depth and normals must be generated, and since many ambient occlusion algorithms work best when using face normals instead of normal-mapped normals, only vertex positions and per-vertex normals are needed to generate the ND-buffer; all other vertex attributes such as texture coordinates and auxiliary textures such as albedo can be ignored. As a result, memory bandwidth consumption is greatly reduced, making the ND-buffer pass considerably cheaper than a full G-buffer pass.

To boost performance even further, we generate an ND-buffer that is a scaling of the original screen size. In doing so, memory bandwidth consumption is reduced by performing fewer texture writes when generating the ND-buffer and by performing fewer texture reads when computing ambient occlusion. In addition, fewer pixels must be processed. As a result, both ND-buffer generation and ambient occlusion become cheaper. From our experiments, we observe that a scaling factor of $\frac{1}{4}$ provides an acceptable trade-off between quality and performance. In addition, this scaling does not affect the resolution of the final image, since the final rendering is done in a separate forward pass.

Once the ND-buffer is generated, it is fed as an input to the ambient occlusion stage. This stage may in fact represent several render passes. For example, many ambient occlusion techniques blur their output to remove noise, amounting to two render passes. And the blurring itself may in turn be implemented in multiple passes, further increasing the complexity of this stage.

Finally, the scene's geometry is rendered once again in a second forward pass. This pass writes directly to the screen's framebuffer and is responsible for computing the scene's illumination and generating the final image that is displayed on screen. In this pass, the ambient occlusion texture generated in the previous stage is used to modulate the ambient lighting term of the lighting equation.

The ND-buffer pipeline offers several advantages that make it a candidate worth considering. First, the ND-buffer pipeline does not require multiple render targets, so it is supported on older OpenGL ES 2.0 hardware; pixel depth can be saved in the `GL_DEPTH_COMPONENT` target of a framebuffer object, and normals can be saved as an RGB texture in the framebuffer's single colour output channel. Given that most OpenGL ES 3.0 devices are middle to top tier at the time of writing, this is a property worth considering if older hardware is to be supported. Second, the ND-buffer pass is much cheaper with respect to a full G-buffer pass, since less data has to be written. This makes this approach especially attractive on low-end devices. Third, since the final output is rendered in a forward pass, the application can make use of hardware multisampling to produce anti-aliased images. Finally, since the ND-buffer and

the final rendering are generated in two separate rendering passes, the ND-buffer can be downscaled for greater performance, while performing the final rendering at full resolution.

The main disadvantage of the ND-buffer pipeline is that geometry must be rendered twice, so an application that is triangle bound will only see this bottleneck increased. Furthermore, the ND-buffer pipeline does not support deferred shading or any rendering technique relying on a full G-buffer containing render targets other than depth and normals. While this may not be an issue on low-end devices, it may be inconvenient if the application targets middle to high-end hardware. In addition, it may sometimes be preferable to store view-space positions instead of depth in the ND-buffer as we will see later, and it is not obvious how to do this without multiple render targets. Finally, the depth value that is written to the `GL_DEPTH_COMPONENT` target of the framebuffer is non-linear, which may produce undesirable results in many ambient occlusion techniques due to the loss of precision in the background.

4.2.2 G-buffer Pipeline

The second pipeline we propose is illustrated in figure 4.2. In this pipeline, we first render the scene's geometry to generate a G-buffer containing three textures: depth or position, normals, and albedo. The depth/position and normal textures are then fed to an ambient occlusion shader in a second pass to produce an ambient occlusion texture. A final compositing pass then takes this texture together with the normal and albedo textures and generates the final output, computing the illumination for every pixel of the screen.

To generate the G-buffer, we render the scene's geometry into a framebuffer object. This framebuffer object contains four render targets: three `GL_COLOR_ATTACHMENT`s storing depth/position, normals, and albedo, and a fourth target containing an auxiliary depth buffer. Since the first three targets must be accessed from shaders in later stages, we attach textures to them. The auxiliary depth buffer, however, need not be sampled, so a renderbuffer is used instead to boost performance.

When generating the G-buffer, we let the application store depth or view-space position depending on which ambient occlusion technique is used. From our experiments, some techniques may run faster when reading view-space positions directly instead of reading depth and then reconstructing positions, so our application has the flexibility to generate one or the other. Since this buffer may contain depth or position, we refer to it as the depth/position buffer.

As with the ND-buffer pipeline, we generate a G-buffer that is $\frac{1}{4}$ of the original screen size. This makes both G-buffer generation and ambient occlusion computation more efficient.

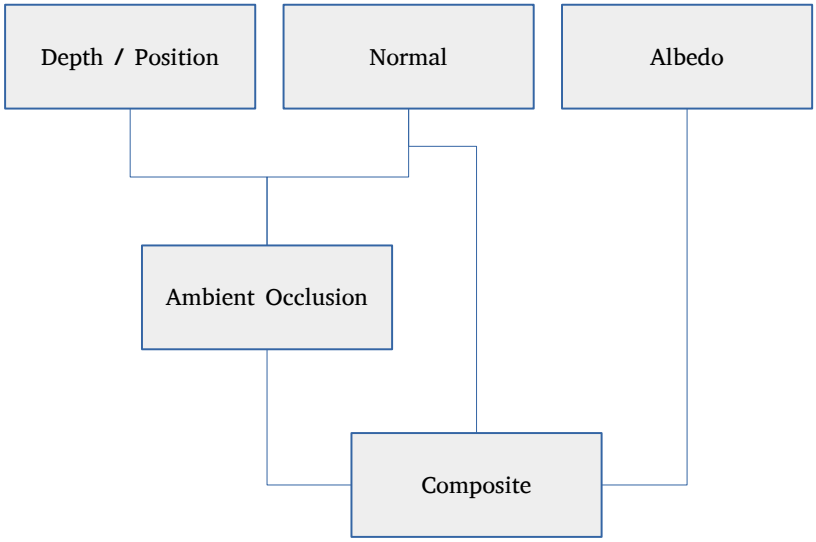


FIGURE 4.2: Ambient occlusion G-buffer rendering pipeline. The scene is rendered once to generate a G-buffer containing depth/position, normals and albedo. The depth/position and normal textures are used in a second stage to compute ambient occlusion. The resulting ambient occlusion texture and the albedo texture are used in a final compositing pass to compute the illumination at every pixel.

Just like in the ND-buffer pipeline, the ambient occlusion stage in figure 4.2 may represent multiple render passes depending on the complexity of the ambient occlusion technique being used.

The final rendering stage does differ from that of the ND-buffer pipeline, however. Like in the ND-buffer pipeline, this stage writes directly to the screen's framebuffer and is responsible for generating the final image that is displayed on screen. Unlike the ND-buffer pipeline, the scene's geometry is rendered only once and not twice; in this final compositing stage, the final output is computed solely from the G-buffer and the ambient occlusion texture generated in the previous stage.

The G-buffer pipeline complements the ND-buffer pipeline in terms of advantages and disadvantages. The G-buffer pipeline allows us to implement deferred shading and other rendering techniques relying on a full G-buffer, and combine them with ambient occlusion. In addition, the scene's geometry need only be rendered once, since the final compositing pass takes its inputs from the G-buffer. Finally, this pipeline allows us to trivially store view-space positions instead of depth, which may sometimes be preferable, as we will see in the results section.

On the other hand, the G-buffer pipeline requires multiple render targets, so this pipeline can only be implemented on OpenGL ES 3.0 compatible hardware or by using non-portable extensions such as `NV_draw_buffers extension` (NVIDIA-only). Furthermore, generating a G-buffer is more expensive than generating an ND-buffer, so

this pipeline is more taxing on memory bandwidth. In addition, since the final compositing takes its inputs from the G-buffer, multisampling is not possible unless extensions like `GL_EXT_framebuffer_multisample` are used. Plus, if the G-buffer is downscaled for performance, so is the final compositing, unless the G-buffer is generated at full resolution and then explicitly downscaled in a separate pass. These two properties combined may potentially result in pixelated images. Finally, this pipeline as is may not be normal-mapping friendly. Many ambient occlusion techniques work best on flat, per-surface normals, and since the normals render target is used for both the ambient occlusion pass and the final compositing, the final compositing would have to compute illumination using flat normals as well. To overcome this, one could either generate a separate normals texture for use in compositing, or simply generate normal-mapped normals instead. The former would increase the tax on memory bandwidth, while the latter would require ambient occlusion shaders to be tweaked.

4.2.3 Pipeline Feature Matrix

For reference and ease of readability, we have summarised the advantages and disadvantages of the two pipelines in what we call the pipeline feature matrix, shown in table 4.1. The two pipelines complement each other in terms of advantages and disadvantages, so the programmer should implement the one that is most suitable for their application.

	ND-buffer	**G-buffer**
OpenGL ES version	2.0+	3.0+ or extensions
Geometry passes	2	1
Memory bandwidth	Low	High
View-space positions	No	Yes
Linear Depth	No	Yes
Multisampling	Hardware	extensions
Downscaling	Free	Impacts final rendering, or additional pass
Deferred Shading	No	Yes
Normal mapping	Yes	May require tweaks

TABLE 4.1: Pipeline feature matrix.

In table 4.1, we have highlighted the characteristics of the pipelines in different colours. Green denotes an optimal or desirable property, while red denotes an undesirable one. Orange denotes a property that may or may not be desirable.

With respect to the OpenGL ES version required by the pipelines, we believe that the fact that the G-buffer pipeline requires a 3.0+ capable device is not entirely undesirable, since it is only a matter of time that old, 2.0 devices phase out in favour of modern, 3.0 ones. According to Unity's hardware statistics page[3], 44.9% of devices are already OpenGL ES 3.0 compatible, so this assumption is not too far from reality.

When discussing downscaling of offscreen buffers, we mentioned that downscaling generally yields a performance boost, but may produce pixelated images in the G-buffer pipeline. On the other hand, rendering at a resolution lower than the screen's native resolution if often desirable in any case, as explained in [Fel15]. For this reason, we believe this property is not entirely undesirable. In a similar manner, we believe the lack of multisampling in the G-buffer pipeline is not a major issue.

Finally, if normal-mapping is to be added to the G-buffer pipeline, ambient occlusion shaders may have to be tweaked. Many algorithms work best when using per-vertex normals or simply face normals rather than normal-mapped normals, so the ambient occlusion shader may have to be modified to prevent over-occlusion. On the other hand, this may reveal subtle occlusion effects due to the normal maps, so this property may or may not be undesirable.

4.2.4 Our Pipeline

In our final implementation, we assume the device supports OpenGL ES 3.0 and implement a G-buffer pipeline. Table 4.2 summarises the characteristics that are specific to our pipeline. We decided on this particular implementation after extensive experimentation and research. The following chapter describes the experiments that led to this result. Finally, note that since our pipeline is a G-buffer pipeline, it inherits all of the properties listed under the *G-buffer* column in table 4.1.

In our rendering pipeline, we generate a G-buffer with three render targets: a 16-bit RGB texture storing view-space positions or a 16-bit R texture storing depth, depending on which ambient occlusion technique is used, an 8-bit RGB texture storing per-vertex normals, and a third 8-bit RGB texture storing albedo.

From our experiments, generating view-space positions instead of depth for some of the ambient occlusion techniques may result in a slight performance boost. Although G-buffer generation is more expensive in this case, the cost is outweighed by these techniques not having to spend computation time reconstructing position from depth. In any case, this may vary from platform to platform, so profiling should be done to decide which approach is best.

Furthermore, given that the G-buffer pipeline allows us to generate linear depth, we choose to do this to avoid depth precision problems in far away geometry.

[3] http://hwstats.unity3d.com/mobile/gpu.html

Pipeline	G-buffer
Render targets	16-bit R[GB] depth/position
	8-bit RGB normals
	8-bit RGB albedo
View-space positions	Depends on shader
Linear Depth	Yes
Multisampling	No
Downscaling	Yes, with impact on final rendering
Z-prepass	No
Normal mapping	No

TABLE 4.2: Characteristics of our pipeline. Note that since this is a G-buffer pipeline, all properties under the *G-buffer* column in table 4.1 are inherited.

As for multisampling and downscaling, we do not perform any multisampling and generate a G-buffer that is $\frac{1}{4}$ of the native screen resolution. The final compositing pass takes its inputs from this G-buffer, and the result is therefore of lower resolution. However, from our experiments, the pixelation resulting from this approach is barely visible, and the savings in terms of computation cost greatly outweigh this quality loss.

In our pipeline, we do not perform a Z-prepass. The cost of the ambient occlusion pass is much greater than that of generating the G-buffer in our tests, so there is little to be gained from a Z-prepass. Again, profiling should be done to determine which approach is best; a Z-prepass might yield a performance boost in scenes with great depth complexity.

Finally, we leave normal mapping aside in our implementation. However, normal maps could easily be added in. The only downside, as previously mentioned, is that ambient occlusion shaders may have to be modified to mitigate over-occlusion.

4.3 Random Sampling

Screen space ambient occlusion techniques approximate the ambient occlusion equation at a point by taking a discrete number of samples around it. Some techniques do this in 3D, sampling a normal-oriented hemisphere centred at the point. Others work in 2D, sampling a disk centred at the projection of the point. In this section, we describe the two sampling methods used in our implementation: disc sampling and hemisphere sampling.

In both sampling methods, we keep the sample count to just eight samples per pixel. Desktop implementations typically gather 16-32 samples, but this range is prohibitive on mobile devices. A sample count of eight yields a reasonable trade-off between quality and performance, reducing memory bandwidth and compute power requirements while delivering acceptable results in real-time.

When using such a low number of samples, the sampling pattern must be chosen carefully. Unlike a desktop implementation, where 16-32 samples are typically used, our implementation cannot rely on a pseudo-random number generator to directly generate sample points. With just 8 samples per pixel, this approach would result in skewed distributions, delivering suboptimal results in the ambient occlusion computation. For this reason, we compute sampling patterns offline and hard-code them into our implementation. This enhances both the robustness and reproducibility of our sampling schemes.

4.3.1 Disc Sampling

Ambient occlusion methods working in 2D sample a local neighbourhood in texture space around the point being shaded. A popular technique to this end, and the one we use in our implementation, is the use of Possion discs. Poisson disc sampling produces sets of points that are closely packed together, but no closer than a given minimum distance, as illustrated in figure 4.3. This is a desirable property in screen space ambient occlusion, where samples should preferably be evenly distributed around the point being shaded. In fact, this property is especially crucial in our implementation, where only a few number of samples are gathered (8). Using a Poisson disc ensures that the distribution of samples is evenly spaced, and results in better quality sampling than just generating random points in the plane with a pseudo random number generator.

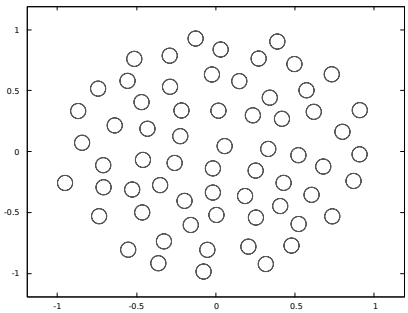


FIGURE 4.3: Poisson disc with 64 samples.

In our implementation, we use the Poisson disc illustrated in figure 4.4. This is a disc of 8 samples that is computed offline using Coderhaus' *Poisson Disk Generator* tool. Coderhaus' tool is an implementation of [Bri07], a modified version of the dart throwing algorithm that generates Poisson disc samples in $\mathcal{O}(N)$ time and in arbitrary dimensions.

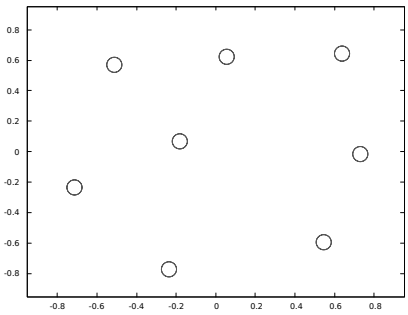


FIGURE 4.4: Poisson disc with 8 samples.

The Poisson disc in figure 4.4 is not used as is in our implementation, however. Every ambient occlusion technique scales this disc to an appropriate size, which is why we conveniently define the samples to be within the unit circle. In addition, all of our ambient occlusion methods apply a random, per-pixel rotation to this disc to avoid banding artifacts.

4.3.2 Hemisphere Sampling

Ambient occlusion methods working in 3D often sample a neighbourhood of points in the normal-oriented hemisphere centred at the point being shaded. Furthermore, in many of these techniques, the occlusion factor is also a function of the cosine of the angle between the normal and the vector from the occluder to the occludee. For this reason, it is convenient to sample the hemisphere according to a cosine-weighted distribution, with more samples around the normal than at grazing angles. Such a distribution is illustrated in figure 4.5.

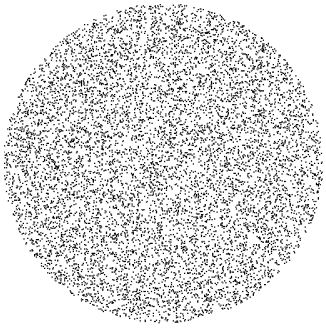


FIGURE 4.5: An example cosine-weighted hemisphere distribution. The figure shows the hemispherical samples projected onto the plane.

In our implementation, we use *Malley's* method to generate cosine-weighted hemisphere distributions as described in [PH10]. The idea behind this approach is to choose points uniformly from the unit disk and then map them to the unit hemisphere. The resulting points on the hemisphere follow a cosine distribution. In our implementation,

we use Poisson disk sampling to generate points on the unit disk, and then map these to the unit hemisphere.

As with the disc-sampling approach, the hemispherical kernel is scaled appropriately by each ambient occlusion method and randomised on a per-pixel basis to remove banding artifacts.

4.3.3 Under-sampling and Per-Pixel Randomisation

Taking only a few number of samples per pixel allows us to approximate ambient occlusion in real-time, but the results suffer from under-sampling. This under-sampling manifests itself visually in the form of banding artifacts, as illustrated in figure 4.6.

FIGURE 4.6: Banding artifacts in Starcraft 2 ambient occlusion due to undersampling.

To remove banding artifacts, we randomly rotate the sample kernel per pixel as described in [Cha11]. We generate a random set of 16 rotation vectors, stored as a 4×4 texture that is tiled all across the screen. Every pixel fetches the rotation vector assigned to it and orients the sample kernel using that vector. We generate 2D rotation vectors for 2D ambient occlusion techniques, and 3D rotation vectors for 3D techniques. Figure 4.7 shows the result of applying this randomisation strategy in Starcraft 2 ambient occlusion.

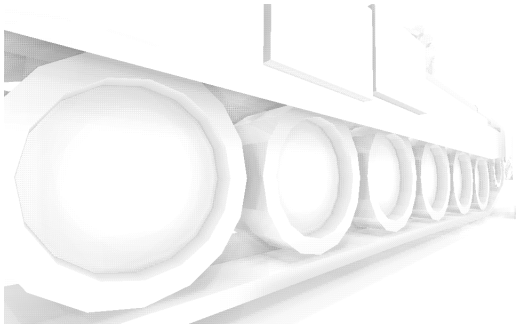


FIGURE 4.7: Noise due to random (under)sampling in Starcraft 2 ambient occlusion.

4.3.4 Blur

The results in figure 4.7 are far from perfect; randomly rotating the kernel per pixel introduces noise in the image. In our implementation, we remove noise using two of the most popular blur filters that are typically used in screen space ambient occlusion: bilateral filter and separable blur. We use the more affordable separable blur by default, and save the bilateral filter for higher-end devices. Figure 4.8 shows the result of blurring the image in figure 4.7 with a separable blur.

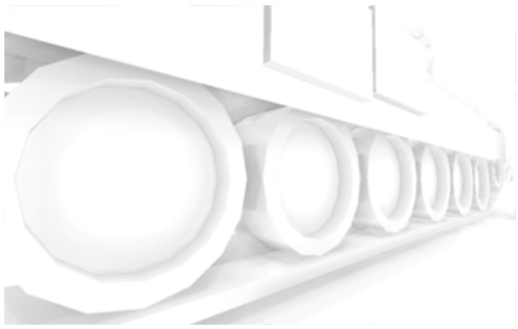


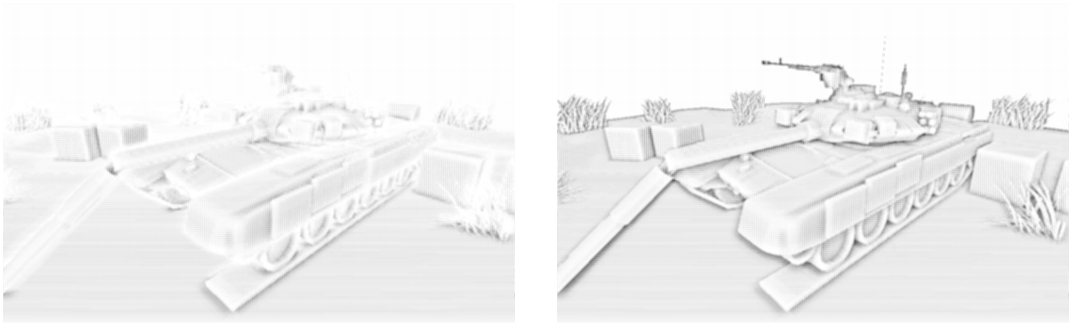
FIGURE 4.8: Blur in Starcraft 2 ambient occlusion.

To determine the dimensions of the blur filter kernels, both quality and performance must be taken into account. A general guideline is to define the blur kernel to have the same size as the rotation texture. We follow this guideline in our implementation, and define a blur kernel size of 4×4 pixels.

4.4 Crytek Ambient Occlusion

Our implementation of Crytek ambient occlusion differs from the original in two points. First, we disable the range check that avoids taking samples from far away geometry. As illustrated in figure 4.9, this results in edge detection, but removes the white halos along the silhouettes of objects and results in less noticeable artifacts along flat surfaces. We believe this approach results in a more artistic and visually pleasing look. Second, since half of the samples in Crytek ambient occlusion are deemed to be occluded for planar surfaces, we remap the occlusion factor from $[0, \frac{1}{2}]$ to $[0, 1]$. The final result can be seen in figure 4.9b.

To generate the sample kernel, we take the simplistic approach of using a pseudo-random number generator to directly generate directions on the unit sphere as described in [Cha11]. We experiment with different seeds until a reasonable pattern is produced. The vectors are then scaled so that most of the samples are taken close to the point being shaded, as described in that same reference. The kernel that we use in our implementation is the 8-point pattern shown in figure 4.10.



(A) With range check (B) Without range check.

FIGURE 4.9: Crytek ambient occlusion with and without range check.

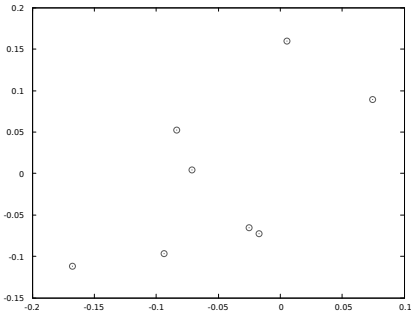


FIGURE 4.10: Crytek sample kernel.

To remove banding artifacts, we randomly rotate the sample kernel per pixel as described in 4.3.3. If we take a patch of 4×4 pixels and draw their rotated kernels in a single image, we obtain the image shown in figure 4.11.

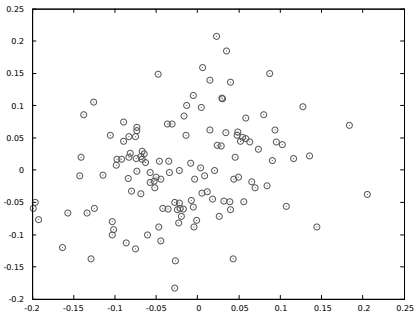


FIGURE 4.11: Crytek sample kernel with per-pixel rotations.

Finally, we remove the noise introduced by random sampling by applying a separable blur filter as described in section 4.3.4.

4.5 Starcraft 2 Ambient Occlusion

In our implementation of Starcraft 2 ambient occlusion, we weigh the occlusion factor with the cosine of the angle between the occluder and the occludee, inspired by the real

ambient occlusion equation. In addition, we use a linear fall-off function to obscure points based on the Z-distance from the occluder to the occludee, with a range check on this Z-distance to avoid gathering samples from far-away geometry. Figure 4.12 shows Starcraft 2 ambient occlusion in our test scene.

FIGURE 4.12: Our implementation of Starcraft 2 ambient occlusion.

To generate the sample kernel, we use the approach described in section 4.3.2. We take the 8-point Poisson disc in figure 4.4 and map it to the unit hemisphere. This results in the pattern shown in figure 4.13, where the x- and y-coordinates of the hemispherical points have been projected onto the plane and then plotted.

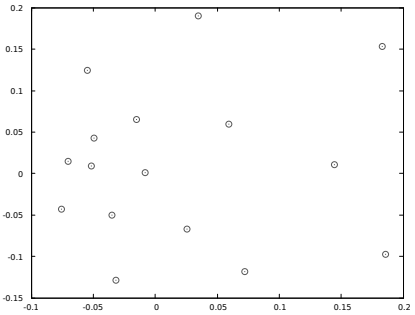


FIGURE 4.13: Starcraft 2 sample kernel.

To remove banding artifacts, we randomly rotate the sample kernel per pixel as described in 4.3.3. If we take a patch of 4×4 pixels and draw their rotated kernels in a single image, we obtain the image shown in figure 4.14.

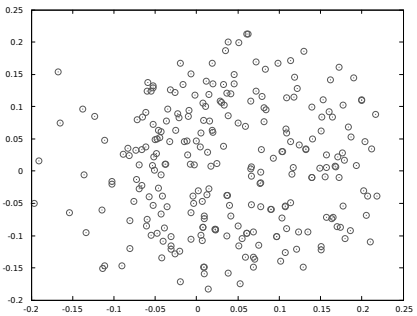


FIGURE 4.14: Stracraft sample kernel with per-pixel rotations.

Finally, we remove the noise introduced by random sampling by applying a separable blur filter as described in section 4.3.4.

4.6 Alchemy Ambient Obscurance

Our implementation of Alchemy ambient obscurance is fairly straightforward. We implement the technique as described in the original paper and tune the algorithm's parameters for our particular application. Figure 4.15 shows our implementation of Alchemy ambient obscurance.

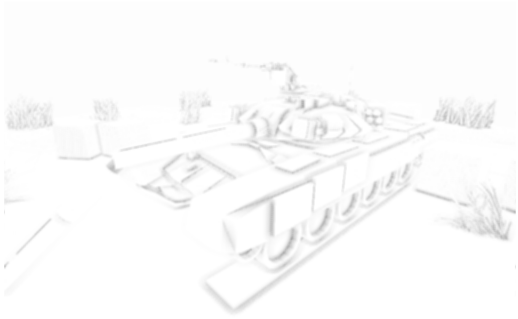


FIGURE 4.15: Our implementation of Alchemy ambient obscurance.

The sample kernel we use in our implementation is the 8-point Poisson disc shown in figure 4.16. This is the same pattern shown in figure 4.4 of section 4.3.1.

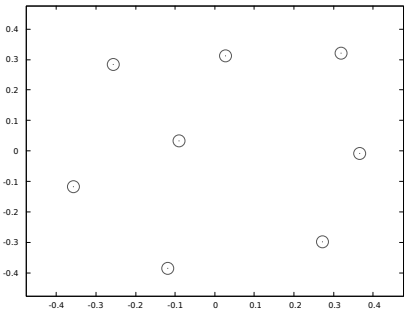


FIGURE 4.16: Alchemy sample kernel.

To remove banding artifacts, we randomly rotate the sample kernel per pixel as described in 4.3.3. If we take a patch of 4×4 pixels and draw their rotated kernels in a single image, we obtain the image shown in figure 4.17.

Finally, we remove the noise introduced by random sampling by applying a separable blur filter as described in section 4.3.4.

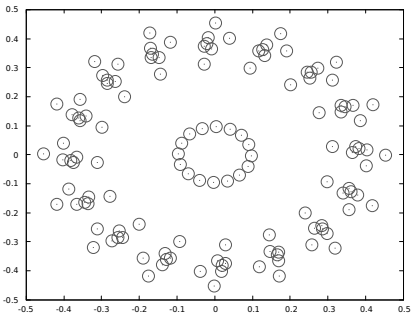


FIGURE 4.17: Alchemy sample kernel with per-pixel rotations.

4.7 Horizon-Based Ambient Occlusion

We borrow the implementation of horizon-based ambient occlusion from [Aal13] and apply a slight modification to the sampling scheme used by the original algorithm. Figure 4.18 shows our implementation of horizon-based ambient occlusion.

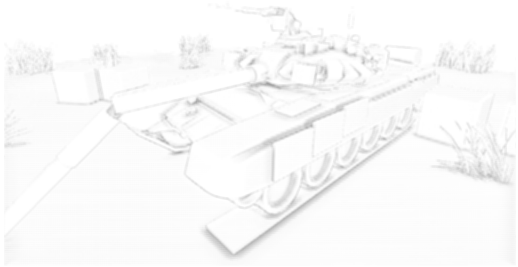


FIGURE 4.18: Our implementation of horizon-based ambient occlusion.

In our implementation, we use the sample kernel illustrated in figure 4.19. Instead of the typical 4×4 sampling scheme — four samples along four rays — we use a 2×4 scheme — two samples along four rays. This keeps the sample count down the eight, resulting in a relatively good balance between quality and performance and allowing us to compare this technique with the other methods.

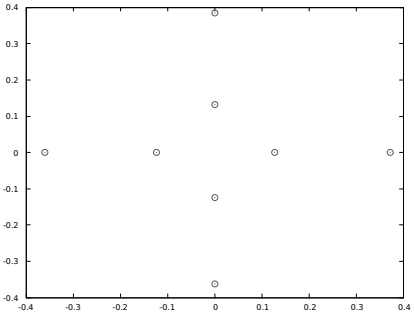


FIGURE 4.19: Horizon-based ambient occlusion sample kernel.

To remove banding artifacts, we randomly rotate the sample kernel per pixel as described in section 4.3.3. However, rotating the kernel with rotation vectors of equal magnitude would produce two perfect circles, so in addition, we jitter the length of the rotation vectors using Gaussian noise to randomise the pattern. If we take a patch of 4×4 pixels and draw their rotated kernels in a single image, we obtain the image shown in figure 4.20.

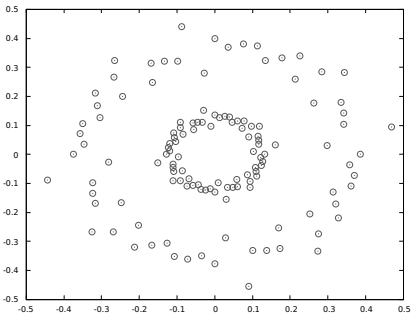


FIGURE 4.20: Horizon-based ambient occlusion sample kernel with per-pixel rotations.

Finally, we remove the noise introduced by random sampling by applying a separable blur filter as described in section 4.3.4.

4.8 Unsharp Mask

In our implementation of unsharp masking of the depth buffer, we compute the high frequency depth buffer by subtracting a blurred version of the buffer from the original, as described in the paper:

$$\Delta D = G * D - D$$

We use a Gaussian blur to blur the depth buffer. For the unsharp mask to produce noticeable effects, a relatively wide kernel must be used, but the kernel must also be kept reasonably small to deliver performance. We use a kernel size of 4×4 in our implementation.

Finally, we subtract a scaling λ of the high frequency depth buffer from the image to darken depth discontinuities, adjusting λ for our particular scene:

$$I' = I + \lambda \Delta D$$

Figure 4.21 shows a side-by-side comparison of our test scene with unsharp masking on and off.

(A) Unsharp mask off. (B) Unsharp mask on.

FIGURE 4.21: Scene rendered with and without unsharp masking of the depth buffer.

4.9 Home-Brewed Ambient Occlusion

As part of our work, we propose a relatively inexpensive ambient occlusion method based on the other techniques. We call this method *home-brewed ambient occlusion*. Like our implementation of Alchemy and other 2D methods, home-brewed ambient occlusion uses the 8-point Poisson disc illustrated in figure 4.4 of section 4.3.1, making its memory access patterns controllable and coherent. In addition, home-brewed ambient occlusion does not rely on any projection or unprojection of samples from view space to texture space and vice versa, giving it a computational advantage over other methods. As a result, home-brewed ambient occlusion is mobile-friendly and relatively simple to implement. Figure 4.22 shows home-brewed ambient occlusion applied to our test scene.

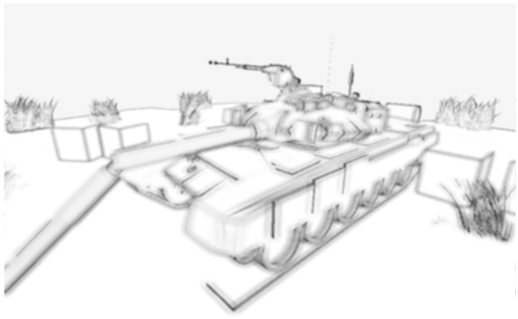


FIGURE 4.22: Home-brewed ambient occlusion.

Home-brewed ambient occlusion defines the occlusion at a point as

$$AO(p) = 1 - \frac{1}{N}\sum_{i=1}^{N} f(d_i)$$

where d_i is the depth difference between the occluder and the occludee and f is the fall-off function:

$$f(d_i) = w_i * \text{smoothstep}(0, 1, (1 + d_i)^2)$$

where w_i is the dot product between the occludee's normal N_p and the occluders's normal N_q:

$$w_i = N_p \cdot N_q$$

The smoothstep term penalises large depth discontinuities, while the w_i term prevents self-occlusion by making the $f(d_i)$ term close to zero if the sample is taken from the same surface as the point being shaded (or, consequently, a surface that is parallel to the surface of the point).

As with other 2D approaches, the sample kernel that we use in home-brewed ambient occlusion is the 8-point Poisson disc shown in figure 4.23. This is the same pattern shown in figure 4.4 of section 4.3.1.

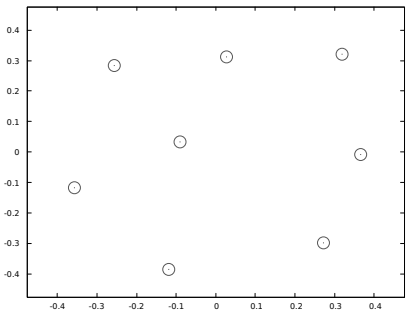


FIGURE 4.23: Home-brewed sample kernel.

To remove banding artifacts, we randomly rotate the sample kernel per pixel as described in 4.3.3. If we take a patch of 4×4 pixels and draw their rotated kernels in a single image, we obtain the image shown in figure 4.24.

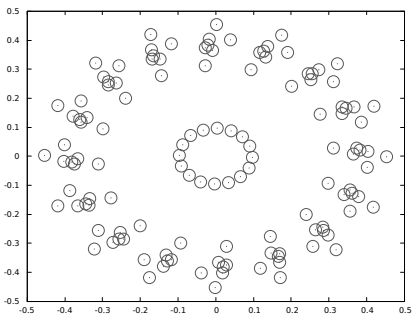


FIGURE 4.24: Home-brewed sample kernel with per-pixel rotations.

Finally, we remove the noise introduced by random sampling by applying a separable blur filter as described in section 4.3.4.

4.10 Progressive Ambient Occlusion

Progressive ambient occlusion is our second proposal. This technique is not an ambient occlusion algorithm per se, but a modification that can be applied to all previous algorithms to speed up their performance. Progressive ambient occlusion can be understood as a simplification of the temporal approach described in [MSW10], or as the separable approach described in [Hua+11], but where the computation is separated in time instead of space.

The idea behind progressive ambient occlusion is to amortise the cost of computing ambient occlusion across multiple frames, gathering only a fraction of the samples in a given frame and progressively refining the ambient occlusion of the scene. If an ambient occlusion algorithm originally takes N samples per pixel, the progressive version of the algorithm would take a fraction M of the original N samples in a given frame and average the result with those of the previous frames. In our implementation, we take $M = 4$ samples per frame for a total of $N = 8$ samples. In frame i, we gather half of the samples and compute an ambient occlusion texture. This texture is averaged with the texture computed using the other half of the samples in frame $i - 1$, and the result is then used as the ambient occlusion term during lighting. This process is illustrated in figure 4.25.

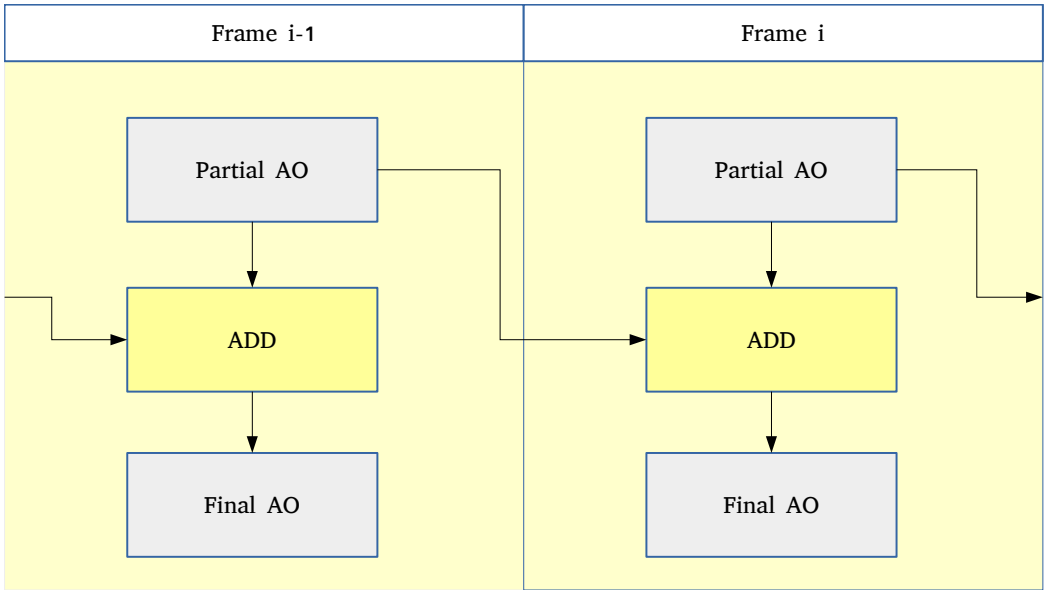


FIGURE 4.25: Schematics of progressive ambient occlusion. An ambient occlusion texture is computed using half of the samples in a given frame, and the result averaged with that of the previous frame. The average is then used to compute the illumination of the current frame. Note that we do not average the current texture with the average of the previous frame, but with the texture computed using the other half of the samples in the previous frame.

Note that we do not average the current ambient occlusion texture with the average

computed in the previous frame, as that would result in a shading effect similar to motion blur. We average the current texture with the texture computed using the other half of the samples in the previous frame; the average of the textures is only used for rendering in the compositing pass.

Chapter 5

Results

In this chapter, we profile our application and discuss the results obtained. We also describe the experiments performed throughout the development of the application that led to the implementation described in the previous chapter.

5.1 Test Setup

In this first section, we describe the test environment that was set up to profile our implementation. This includes the test scene, profiling tools, and test devices used.

To test the different ambient occlusion methods, we have put together the test scene illustrated in figure 5.1. This scene is comprised of $61.9k$ vertices, $66.1k$ triangles, and 6 textures. These three numbers directly influence the cost of the G-buffer and forward rendering passes, so we find it noteworthy to briefly describe our test scene here to give the reader an idea of the complexity of our test scene.

FIGURE 5.1: Our test scene.

In the following sections, we profile our application and show the time spent in each major stage of the rendering pipeline. To do this, we have written a library to time regions of code in an application. Since the GPU and CPU work asynchronously, we make the library issue `glFinish()` calls before and after every stage of the rendering pipeline. In doing so, we force the GPU to flush all of its work and obtain the time spent in that particular stage. Additionally, since issuing `glFinish()` calls at the

start and end of every region and at every frame would make the application run at a very low frame rate, we gather profiling samples stochastically once every given number of frames on average instead of doing so at every frame. In this way, we obtain performance statistics while still being able to interact with the application.

Another issue to consider when profiling screen space ambient occlusion is that performance is often a function of the distance from the camera to the visible scene geometry. For example, Starcraft 2 ambient occlusion samples in view space and then projects the samples to texture space. If the geometry is close by, the projection of the samples occupies a wider area in texture space, making texture fetch operations more incoherent and therefore making the occlusion computation slower. For this reason, we automate the animation of the camera during profiling to avoid user-introduced bias. In this animation, the camera orbits around the tank at a constant radius and velocity.

In the sections that follow, we measure the time spent in each of the stages of the rendering pipeline. The measurements shown are always averages, taken during the automated animation described above across multiple frames. Thus, when we state that a particular shader or pipeline stage takes so much amount of time to run, we are implicitly referring to the average amount of time.

Finally, in terms of devices used, we have developed our application on a Google Nexus 7 (2013). Unless otherwise stated, all of the profiling results that follow correspond to this device. We have also profiled our application on a Google Nexus 5 (2013) and on an NVIDIA Shield Tablet K1 (2015). When we show results for these two devices, we will explicitly state that the results correspond to them and not to the Nexus 7. Table 5.1 shows the specifications of all these devices, and table 5.2 their GPU specifications.

	Nexus 7	**Nexus 5**	**Shield K1**
CPU	Quad-core 1.5 GHz	Quad-core 2.3 GHz 400	Quad-core 2.2 GHz
GPU	Adreno 320	Adreno 330	Tegra K1
RAM	2 GB	2 GB	2 GB
Resolution	1200×1920	1080×1920	1920×1200

TABLE 5.1: Specifications of the devices used to develop and test our implementation.

5.2 Forward and Deferred Pipelines

As mentioned in section 4.2, we propose two different rendering pipelines for the implementation of screen space ambient occlusion on mobile. In this section, we explain

	Adreno 320 (Nexus 7)	**Adreno 330 (Nexus 5)**	**NVIDIA K1 (Shield K1)**
Cores	16	32	192
Core Speed	400 MHz	400-578 MHz	950 MHz

TABLE 5.2: GPU specifications of our test devices.

why we decided to implement the G-buffer pipeline and show the results of our initial experiments.

As discussed in section 4.2, the downsides of the G-buffer pipeline are:

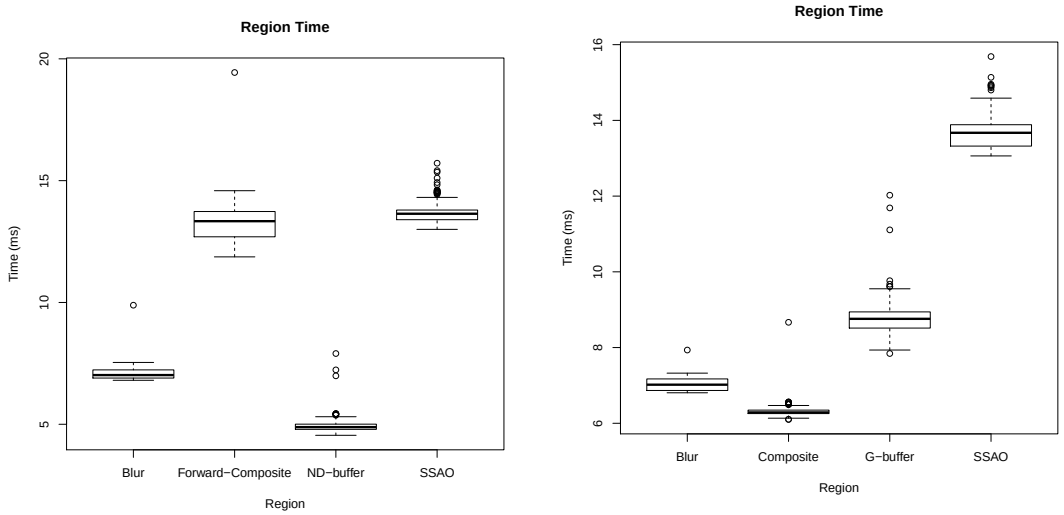
- OpenGL ES 3.0+ required
- High memory bandwidth requirements
- Multisampling available only with extensions
- Downscaling either impacts final rendering or requires an additional pass
- Normal mapping may require tweaks

We believe the OpenGL ES 3.0+ version requirement is not a major concern, since it is only a matter of time that 2.0 devices phase out in favour of the more modern 3.0 ones. In fact, according to Unity's hardware statistics page, ES 3.0 devices already make up 44.9% of the user base at the time of writing[1]. In addition, normal mapping can be used with the G-buffer pipeline; some of the ambient occlusion shaders may just have to be tweaked to prevent over-occlusion. On the other hand, downscaling and the lack of multisampling produce pixelation, but since mobile devices put very high resolutions into small form factors, the result is still acceptable compared to desktop screens. Plus, rendering at non-native resolutions is often preferable, as discussed in [Fel15]. The only major concern, therefore, is the memory bandwidth cost of this pipeline.

To gain further insight into the memory bandwidth cost of the G-buffer pipeline, we profile both the G-buffer and the ND-buffer pipelines and analyse the results. Figure 5.2 shows a performance comparison between the two. On the left, we see how the ND-buffer pipeline is faster to generate (4.9ms) versus the G-buffer pipeline on the right (8.8ms). However, this alone does not justify the use of the ND-buffer pipeline. The ND-buffer pipeline requires a second forward pass, shown as *Forward-Composite* on the left. This forward pass is considerably slower than the compositing pass in the G-buffer pipeline on the right (13.3ms versus 6.3ms). Adding both components together shows that the G-buffer pipeline is 3.1ms faster than the ND-buffer pipeline (4.9 + 13.3 = 18.2ms, 8.8 + 6.3 = 15.1ms, 18.2 - 15.1 = 3.1ms). To add to this difference, note that the second forward pass becomes worse with increasing numbers of triangles, whereas

[1] http://hwstats.unity3d.com/mobile/gpu.html

the performance of the compositing pass in the G-buffer pipeline is only a function of resolution. Given that our scene is relatively simple in terms of triangle count and that resolution remains relatively constant throughout devices, we think the G-buffer pipeline is a better choice than the ND-buffer pipeline.



(A) ND-buffer pipeline with forward final pass. (B) G-buffer pipeline with deferred compositing.

FIGURE 5.2: Performance comparison of the ND-buffer and G-buffer pipelines. Generating an ND-buffer is faster than generating a G-buffer, but the cost of the second forward pass in the ND-buffer pipeline shifts the balance towards the G-buffer pipeline.

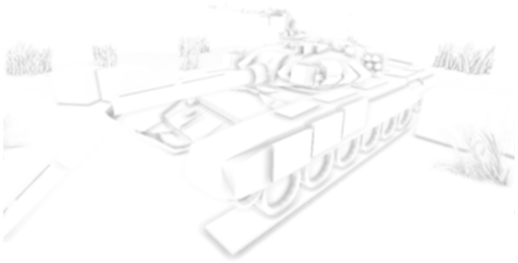
Having profiled the application, we conclude that the G-buffer pipeline is a better choice for us. However, a programmer should always consider their options and profile their application to make the right choice. If the application must support OpenGL ES 2.0 devices, for example, the G-buffer pipeline is simply not an option.

5.3 Depth Precision

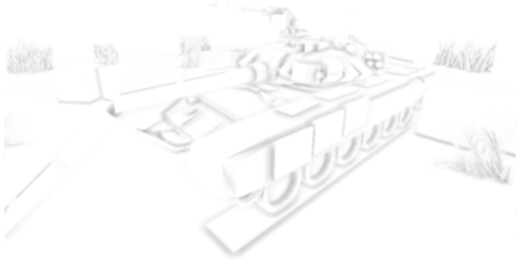
One of the earliest experiments we performed was experimenting with different depth buffer precisions for the depth render target of the application's G-buffer. In this test, we save linear depth using different depth texture formats — `GL_R8` (8 bits), `GL_R16F` (16 bits) and `GL_R32F` (32 bits) — and analyse the quality and performance of the Alchemy ambient obscurance algorithm using each of these formats.

Figure 5.3 shows the Alchemy ambient obscurance algorithm using each of the test formats. As seen in the figure, an 8-bit depth produces unacceptable results. With 16- and 32-bit depths, the ambient occlusion shader produces reasonable results, with the quality difference between 16- and 32-bit being negligible for our test scene.

(A) 8-bit depth (GL_R8).



(B) 16-bit depth (GL_R16F).



(C) 32-bit depth (GL_R32F).

FIGURE 5.3: Alchemy with different depth buffer precisions.

Since a 16-bit depth texture produces good enough results for us, we use this format in our implementation to save memory bandwidth versus a 32-bit depth texture. However, we would still like to understand the performance impact of using this latter one, since scenes with a greater depth complexity might require higher precision.

In figure 5.4 and table 5.3, we show the time spent on different parts of the rendering pipeline as well as the overall frame time using 16- and 32-bit depth buffers. As we can see, a 32-bit depth buffer does have an impact due to higher memory bandwidth requirements on different parts of the pipeline, namely the G-buffer, ambient occlusion and blur passes. However, this impact is only minimal, and would be justified in scenes of great depth complexity where a 16-bit depth buffer would yield errors in the ambient occlusion computation.

Region	**16-bit**	**32-bit**
G-buffer	7.504	7.784
SSAO	17.89	18.02
Blur	6.750	7.438
Frame	34.78	35.77

TABLE 5.3: Time spent in milliseconds on different parts of the pipeline as well as overall frame time using 16-bit and 32-bit depth buffers. (less is better).

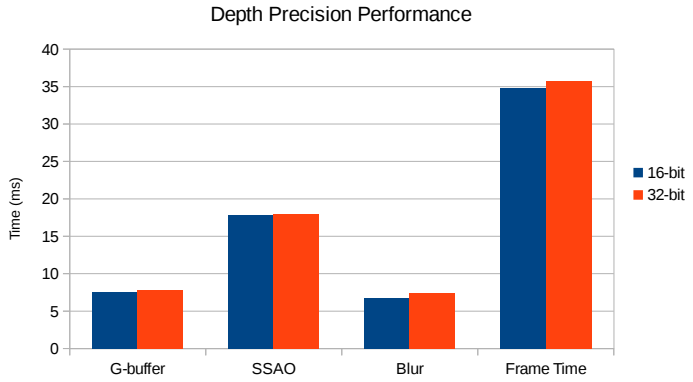


FIGURE 5.4: Time spent on different parts of the pipeline as well as overall frame time using 16-bit and 32-bit depth buffers. (less is better).

In conclusion, we think the programmer should choose the lowest precision for the depth buffer that satisfies their requirements. A lower precision buffer saves memory bandwidth and translates to higher performance in all stages of the pipeline that require reading or writing from/to the depth buffer. Using higher precision has only a minimal impact, however, so the use of a higher precision depth buffer is justified in scenes of great depth complexity.

5.4 View Space Position Reconstruction

In this test, we experiment with two different methods to reconstruct view space position from depth: using the inverse of the projection matrix and using similar triangles[2]. Reconstruction using similar triangles requires less float-point operations, both on the CPU side when inverting the projection matrix and on the GPU side when performing the actual reconstruction. Our intuition was that the similar triangles approach would be fastest.

In terms of quality, both methods produce results with negligible differences. Figure 5.5 shows our implementation of Alchemy ambient obscurance using the inverse of the projection matrix (left) and similar triangles (right).

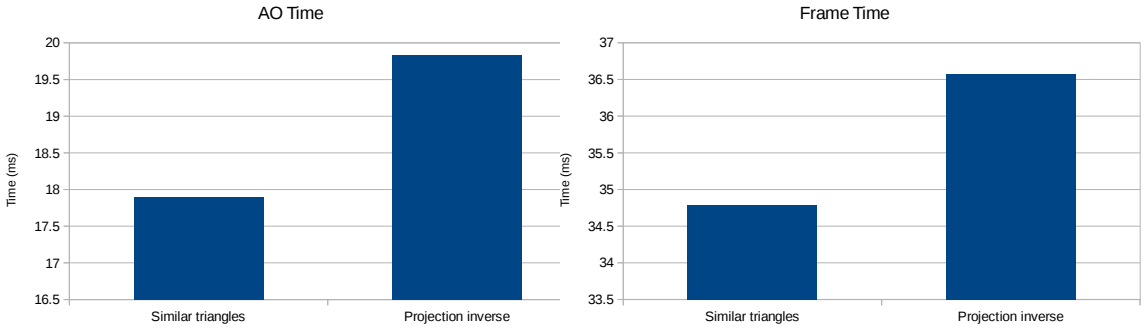
In terms of performance, the similar triangles method is slightly faster than the projection inverse method, as illustrated in figure 5.6. From our experiments, the Alchemy ambient occlusion algorithm runs at 17.9ms using similar triangles and at 19.83ms using the inverse projection method. This translates to an overall frame time of 34.79ms using similar triangles and 36.58ms using the inverse projection.

Since using similar triangles to reconstruct position from depth is faster than the projection inverse method, we use similar triangles in our implementation.

[2] http://shellblade.net/unprojection.html

(A) Using projection matrix inverse. (B) Using similar triangles.

FIGURE 5.5: View-space position reconstruction using (A) the inverse of the projection matrix and (B) similar triangles. The quality difference is negligible.



(A) AO time. (B) Frame time.

FIGURE 5.6: Performance comparison of depth reconstruction using similar triangles and projection inverse (less is better). The left figure shows ambient occlusion time. The right figure shows overall frame time. Reconstruction using similar triangles is slightly faster than the projection inverse method.

5.5 Saving View Space Position Instead of Depth

The next test we performed was saving view space position instead of depth in the G-buffer. Our intuition was that we could speed up ambient occlusion shaders by not having to reconstruct position from depth at the cost of increased memory bandwidth. In this experiment, we run the Alchemy, Crytek, horizon-based ambient occlusion (HBAO) and Starcraft 2 (SC2) shaders with a depth buffer on one hand and a position buffer on the other hand. Since the other ambient occlusion techniques work in 2D space, it makes no sense for them to use a position buffer, so they are excluded from this experiment.

Figure 5.7 and table 5.4 show the frame times of each of the methods using depth and position buffers. From these results, we can see that using a depth buffer is only slightly faster than using a position buffer for most shaders. For the Starcraft 2 shader, the position buffer is actually faster.

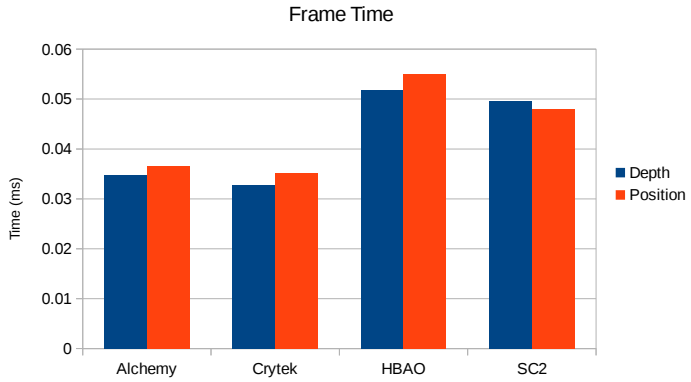


FIGURE 5.7: Frame times of ambient occlusion methods using depth and position buffers. Using a depth buffer is faster for all shaders except for the Starcraft 2 method.

Technique	Depth	Position
Alchemy	0.03478	0.03661
Crytek	0.03283	0.03514
HBAO	0.05177	0.05494
SC2	0.04963	0.04805

TABLE 5.4: Frame times of ambient occlusion methods using depth and positon buffers. Using a depth buffer is faster for all shaders except for the Starcraft 2 method.

To better understand these results, we profile each of the shaders using depth and position buffers. Figure 5.8 shows a profiling session for each shader-buffer configuration pair. Table 5.5 summarises the numerical values behind this figure.

Focusing on the Alchemy ambient obscurance algorithm, we see that, as expected, G-buffer generation is faster when using a depth buffer than when using a position buffer (7.5ms versus 8.9ms). On the other hand, the Alchemy shader is also slightly faster when using a depth buffer than when using a position buffer (17.9ms versus 18.28ms), so the increased memory bandwidth does outweigh the computation savings. Combining both components, we see that using a depth buffer is overall faster (7.5 + 17.9 = 25.4ms, 8.9 + 18.28 = 27.18ms).

The other ambient occlusion methods exhibit a similar behaviour to that of Alchemy, except for the Starcraft 2 shader. For this latter one, using a position buffer is slightly faster than using a depth buffer. Like the other methods, G-buffer generation for Starcraft 2 ambient occlusion is faster when using a depth buffer versus using a position buffer (7.5ms versus 9.0ms). However, the shader itself is slower when using a depth buffer (33.9ms versus 30.6ms). Adding both components together explains why this method is faster with a position buffer (7.5 + 33.9 = 41.4ms, 9.0 + 30.6 = 39.6ms).

In general, we find the results of this experiment to be inconclusive. A depth buffer

Technique/Buffer	**Blur (ms)**	**Composite (ms)**	**G-Buffer (ms)**	textbfSSAO (ms)
Alchemy/Depth	6.730	6.358	7.552	17.89
Alchemy/Position	6.959	6.323	8.919	18.28
Crytek/Depth	6.760	6.344	7.551	15.72
Crytek/Position	6.974	6.341	8.893	16.81
HBAO/Depth	6.725	6.355	7.603	36.34
HBAO/Position	6.920	6.477	8.995	38.42
Starcraft 2/Depth	6.714	6.329	7.561	33.98
Starcraft 2/Position	6.901	6.335	9.039	30.60

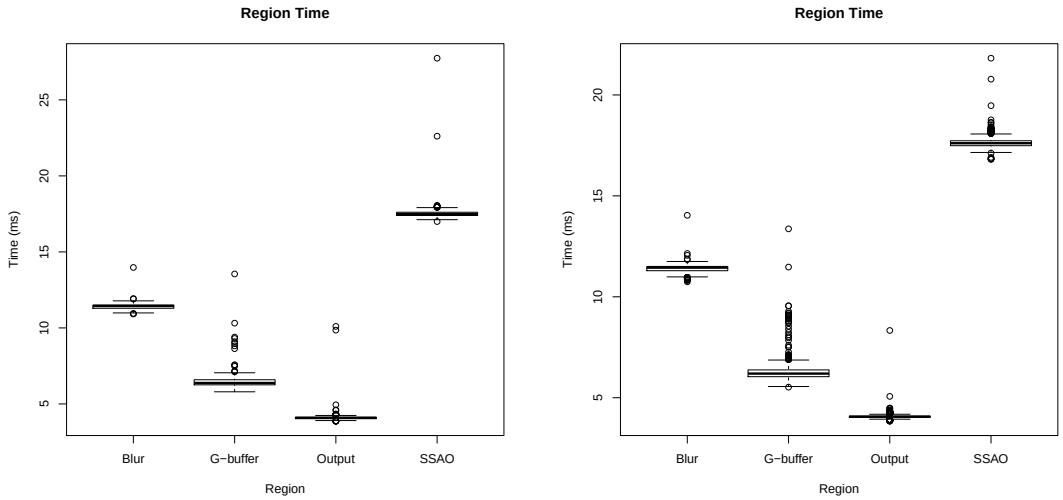
TABLE 5.5: Profiling of each of the ambient occlusion methods using depth and position buffers. The table shows all shader/buffer configuration pairs and the time spent in milliseconds in each stage of the pipeline.

appears to be slightly faster than a position buffer, but the difference is negligible and the results are not consistent among all ambient occlusion methods. We think the programmer should profile their application on their target platform and decide which of the two approaches delivers the best performance. It is for this same reason that we support both depth and position buffers in our pipeline.

5.6 Saving Normals as RG

In this experiment, we experiment with saving normals into two channels instead of three. Originally, we would write the `x`, `y`, and `z` components of the normals in an 8-bit RGB texture. In this experiment, we write `x` and `y` in an 8-bit RG texture and reconstruct `z` in shaders. Since normals are normalised, $|N| = \sqrt{x^2 + y^2 + z^2} = 1$, and therefore $z = \sqrt{1 - x^2 - y^2}$. This allows us to save just two channels (`x` and `y`) and reconstruct the third (`z`).

Figure 5.9 shows the results of this experiment. On the left, we save the `xyz` components of the normals in an RGB texture and then run the Alchemy shader to compute ambient occlusion. On the right, we save the `xy` components of the normals in an RG texture and reconstruct the `z` component in the shader. The savings are negligible. The G-buffer pass is somewhat faster (from 6.4ms to 6.2ms), but the ambient occlusion pass becomes more expensive (from 17.5ms to 17.6ms). Also note that Alchemy only accesses the normal texture once per fragment. Other techniques access it several times per fragment, in which case the performance of the ambient occlusion pass would only become worse.



(A) RGB normals. (B) RG normals.

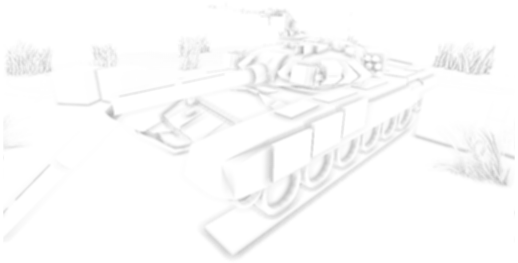
FIGURE 5.9: Alchemy shader performance using a (A) RGB normals (B) RG normals.

In our pipeline, we simply write normals into an RGB texture, since this approach appears to be faster overall.

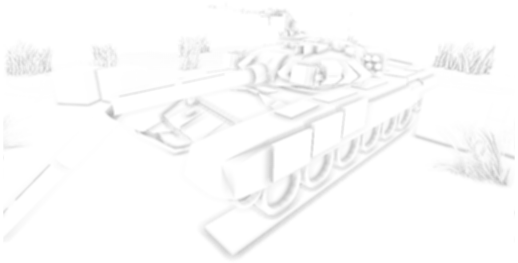
5.7 Bilateral Filter and Separable Blur

In this test, we benchmark the bilateral filter method against the separable blur. We run our implementation of Alchemy ambient obscurance with both blur methods and measure their performance.

First, as illustrated in figure 5.10, we note that the quality difference between the bilateral filter and the separable blur is negligible. This is especially true on mobile, where a small form factor is combined with a high screen resolution.



(A) Alchemy using bilateral filter.



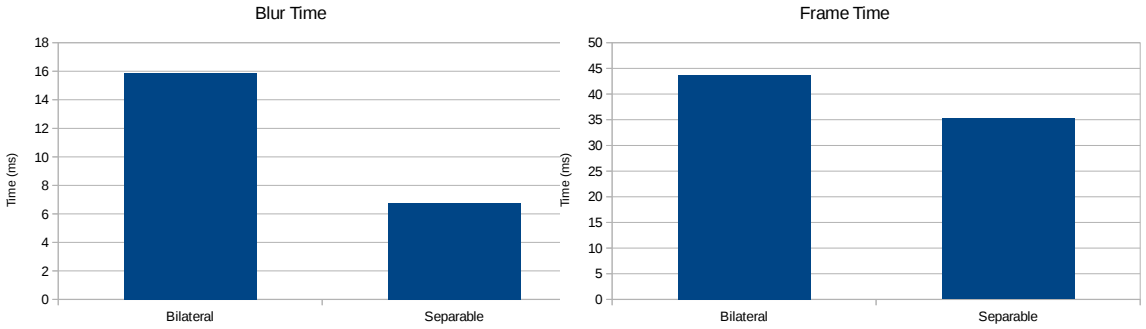
(B) Alchemy using separable blur.

(C) Alchemy using bilateral filter.

(D) Alchemy using separable blur.

FIGURE 5.10: Quality comparison between bilateral filter (A,C) and separable blur (B,D). Although a difference does exist, we find it to be negligible, especially on small form factors such as mobile.

Next, we measure the performance of both blur methods. Figure 5.11 shows the results of this experiment. On the left, we show the time spent by each of the blur methods — 15.87ms by the bilateral filter, and 6.76ms by the separable blur. On the right, we show the frame time of the entire rendering pipeline when running the Alchemy ambient obscurance algorithm with each blur method. Alchemy runs at 43.7ms per frame when using a bilateral filter, and at a considerably faster 35.2ms per frame when using a separable blur.



(A) Blur time.

(B) Frame time.

FIGURE 5.11: Performance measurements of bilateral filter and separable blur in the Alchemy ambient obscurance algorithm (less is better). The separable blur offers a considerable boost in performance on our target platform.

From this experiment, we conclude that a separable blur is faster than a bilateral filter on our target mobile platform, despite the former requiring one additional render pass.

5.8 Runtime Performance of Ambient Occlusion Methods

After performing all of the above optimisations, we benchmark each of the screen space ambient occlusion methods that we have implemented. In each test, we generate a G-buffer, invoke the ambient occlusion shader, blur the result with a separable blur if necessary and combine the results with the albedo texture in a final compositing pass.

The results of this benchmark are shown in figures 5.12 and 5.13 and table 5.6. Figure 5.12 shows the time taken by each ambient occlusion shader, omitting G-buffer generation, blur and compositing. Figure 5.13, on the other hand, shows the frame times obtained by each of the methods. Table 5.6 shows the numbers behind these two figures. From the results, we observe that home-brewed ambient occlusion is the fastest method, followed by Crytek, Alchemy, Starcraft 2, horizon-based ambient occlusion and the unsharp mask method.

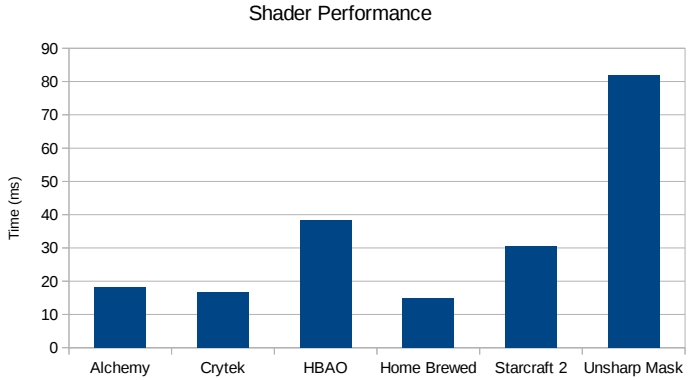


FIGURE 5.12: Shader time of each of the ambient occlusion methods (Nexus 7, less is better).

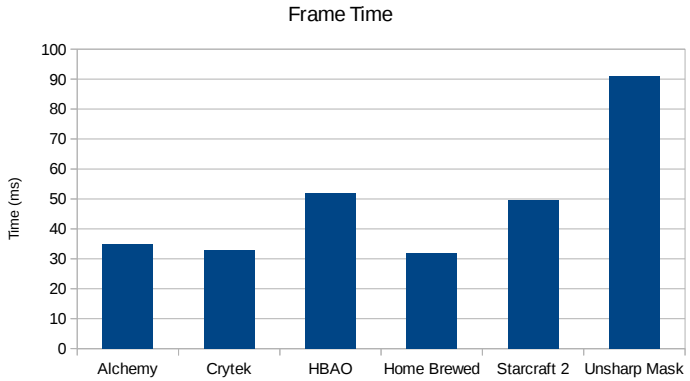


FIGURE 5.13: Frame time of each of the ambient occlusion methods (Nexus 7, less is better).

Method	**Shader Time**	**Frame Time**
Alchemy	18.26	34.79
Home Brewed	14.76	31.98
Crytek	16.77	32.8
Starcraft 2	30.58	49.62
HBAO	38.25	51.78
Unsharp Mask	81.78	91.18

TABLE 5.6: Shader and frame times of each of the ambient occlusion methods.

Figures 5.14 and 5.15 show the results of this same experiment on a Nexus 5 device. Here, we observe that the trend is similar, with Crytek, home-brewed and Alchemy being the fastest methods, but in a different order. Overall, the shader and frame times are smaller than on the Nexus 7 device, since the Nexus 5 has a more powerful GPU. Also note that in figure 5.15, the frame time of the Crytek, home-brewed, and Alchemy methods appear to be the same. This is because the device has v-sync enabled, in which case the frame time cannot be smaller than 16.6 ms (a framerate greater than 60 fps).

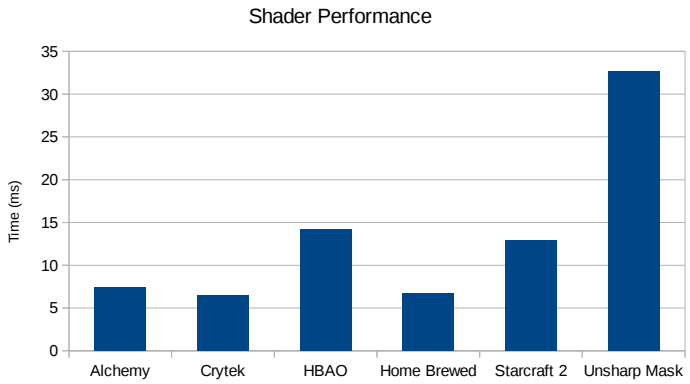


FIGURE 5.14: Shader time of each of the ambient occlusion methods (Nexus 5, less is better).

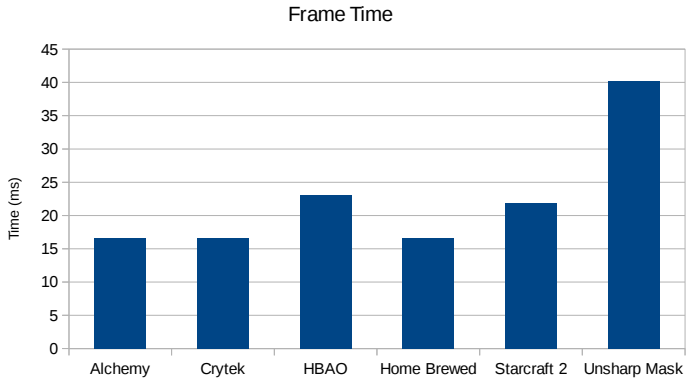


FIGURE 5.15: Frame time of each of the ambient occlusion methods (Nexus 5, less is better).

Similarly, figures 5.16 and 5.17 show the shader times and frame times, respectively, on an NVIDIA Shield K1. Again, the trend repeats itself, albeit in a new order. Here, Alchemy is the fastest method, followed by Crytek and home-brewed. In addition, the frame times for each of these methods are again the same due to v-sync.

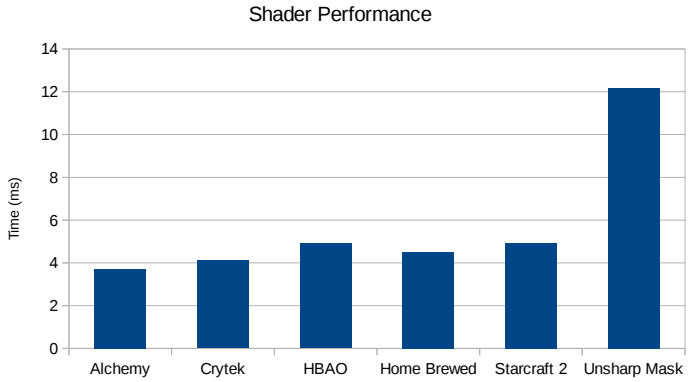


FIGURE 5.16: Shader time of each of the ambient occlusion methods (NVIDIA Shield K1, less is better).

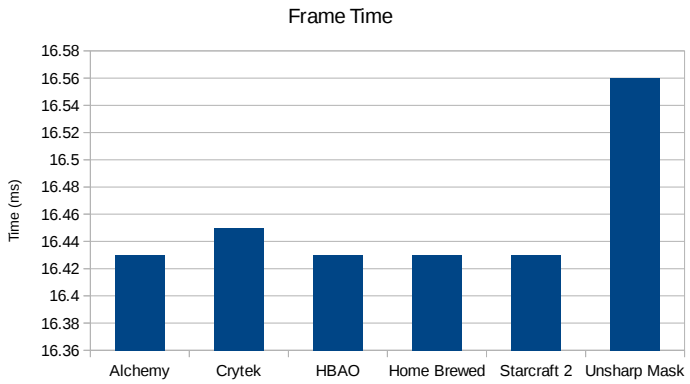


FIGURE 5.17: Frame time of each of the ambient occlusion methods (NVIDIA Shield K1, less is better).

Method	Depth	Normal	Sampling	Projections	Unprojections
Alchemy	N+1	1	2D	0	N+1
Home Brewed	N+1	N+1	2D	0	0
Crytek	N+1	0	3D	N	N+1
Starcraft 2	N+1	1	3D	N	N+1
HBAO	1 + R(2 + S)	0	3D	0	1 + R(2 + S)

TABLE 5.7: Ambient occlusion methods listed from fastest to slowest (the first three swap in ranking based on platform, here we show only one of the orderings). For each method, we list the number of depth texture fetches, normal texture fetches, whether samples are gathered in texture space (2D) or view space (3D), the number of projections from view space to texture space and the number of unprojections from texture space to view space. N refers to the number of samples. For HBAO, R is the number of rays and S the number of samples per ray.

From all of these results, we can conclude that Alchemy, home-brewed and Crytek are the fastest methods, although they seem to swap positions in the ranking based on the target platform. On the other hand, HBAO, Starcraft 2 and the unsharp mask method are the slowest.

From the above results, it appears that the unsharp mask method consistently ranks as the slowest method. Like the bilateral filter, the unsharp mask algorithm gathers a square number of samples, and we believe this is the reason of its slow performance. Comparing the unsharp mask with the other ambient occlusion methods is similar to comparing the bilateral filter with the separable blur: an area kernel is too expensive on mobile, and this is what makes the algorithm prohibitively slow.

To gain further insight on the performance of the other ambient occlusion methods, we have noted down some of their key characteristics in table 5.7. In this table, we show each of the methods ordered from fastest to slowest. Note that the first three methods swap in ranking order based on platform, so we show only one of the orderings in this table. For each method, we note down the number of depth and normal texture fetches, whether the method gathers samples in view space (3D) or directly in texture space (2D), as well as the number of projections from view space to texture space and unprojections from texture space to view space performed. In addition, we use colour keying to denote positive or negative aspects of the algorithms. Green denotes a positive aspect, for example, a few number of normal texture fetches, and red a negative aspect. The names of the methods themselves are also coloured: green denotes a fast method, and red a slow method.

Several conclusions can be drawn from table 5.7. First, note that even though we cannot assert that 2D methods (those that sample directly in texture space) are the fastest, since Crytek is the fastest method on the Nexus 5, what we can assert is that they are always

among the fastest. We believe this is due to their sampling nature. For these methods, we compute a Poisson disc offline that is then used at runtime to randomly sample the depth and normal buffers. This approach guarantees that accesses to the depth and normal textures are coherent, optimising the memory access patterns of these 2D methods. Given that mobile devices offer very limited memory bandwidth, this leaves this set of methods at an advantage with respect to the 3D ones.

The second observation we note is that the three fastest methods — Alchemy, home-brewed, and Crytek — are all computationally inexpensive. On the other hand, Stracraft 2 and HBAO are both relatively expensive in terms of the number of floating-point operations performed. In this way, it seems mobile platforms and their limited compute power reward those shaders that perform few computations.

The third and final observation that can be drawn from this table is that the three fastest methods all offer a trade-off between memory savings and compute savings. For example, the Alchemy shader performs $N+1$ unprojections (computationally expensive) but only accesses the normal texture once (memory savings). On the other hand, the home-brewed algorithm accesses the normal texture $N+1$ times (memory expensive) but performs no projections or unprojections (computational savings). Finally, the Crytek shader performs N projections and $N+1$ unprojections (computationally expensive) but does not access the normal texture at all (memory savings). We believe this is what causes these three shaders to swap in ranking order in different platforms. On some of these platforms, the memory savings outweigh the computational ones, whereas other platforms exhibit the opposite behaviour. As a result, it is unclear which of the three algorithms is the fastest, and only profiling can tell whether the platform rewards memory savings over computational ones or vice versa.

5.9 Progressive Ambient Occlusion

In this section, we profile each of the ambient occlusion shaders for which a progressive implementation makes sense (all but the unsharp mask method) and compare their performance with their original, non-progressive counterparts.

Figure 5.18 shows the frame times delivered by each of the ambient occlusion methods with and without the progressive approach. As can be seen in the figure, the progressive approach significantly speeds up all of the ambient occlusion methods, even though an extra pass is required to average the results of the current frame's ambient occlusion with that of the previous frame.

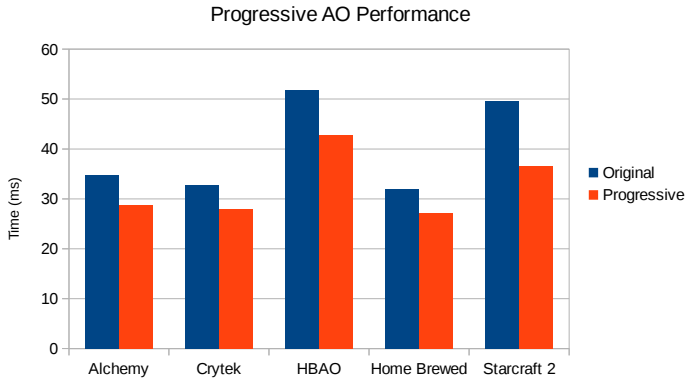


FIGURE 5.18: Frame time of each of the ambient occlusion methods with and without the progressive approach (less is better).

We can gain further insight into the progressive ambient occlusion approach by profiling each of the methods. Figure 5.19 and table 5.8 show the time spent in each of the ambient occlusion shaders with and without the progressive approach. Note that when we measure shader time in the progressive case, this also includes the time spent averaging one frame's ambient occlusion with the previous frame's. In this way, the shader time of the Alchemy algorithm is in fact 11.55ms - 1.094ms = 10.46ms.

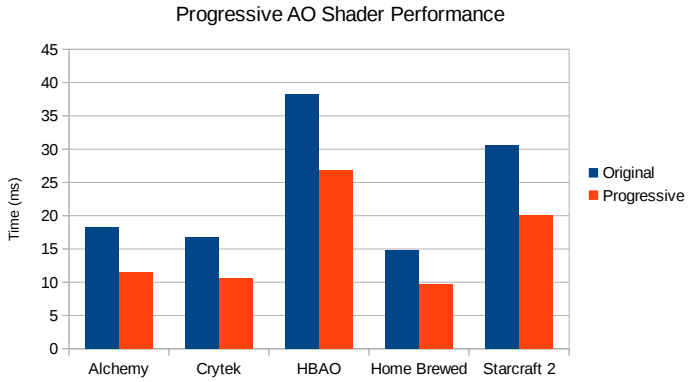


FIGURE 5.19: Shader time of each of the ambient occlusion methods with and without the progressive approach (less is better).

Method	Original	Progressive	Average
Alchemy	18.26	11.55	1.094
Crytek	16.77	10.64	1.071
HBAO	38.25	26.81	1.055
Home Brewed	14.76	9.75	1.059
Starcraft 2	30.58	19.99	1.016

TABLE 5.8: Shader time of each of the ambient occlusion methods with and without the progressive approach (less is better).

Figure 5.20 shows the results of profiling each of the ambient occlusion methods using the progressive approach. In these results we introduce one new region, labelled *PAVG*, which refers to the average done to add the contribution of the previous frame's ambient occlusion with that of the current frame's. Again, when we measure the time spent in the ambient occlusion shader (*SSAO*), this also includes the time spent computing the average (*PAVG*). In all cases, we see how the ambient occlusion shader sees its performance boosted at the cost of a 1.1ms average. Overall, the progressive approach yields faster ambient occlusion computations, despite the added cost of the average pass.

To conclude this section, it is worth noting that the progressive approach does not come for free. Even though performance is indeed boosted, progressive ambient occlusion introduces some minor flickering when the camera is animated. This flickering is exaggerated the lower the framerate is. Since we use the previous frame's ambient occlusion in the current frame's, the lower the framerate, the greater the time between both ambient occlusion computations, and the more incorrect the progressive approach becomes. As a consequence, flickering is introduced.

5.10 Qualitative Results and Comparison

In this section, we provide a qualitative comparison among all our ambient occlusion implementations.

5.10.1 Crytek Ambient Occlusion

Crytek ambient occlusion is the second fastest method from our results. However, this speed comes at a cost: the approach effectively throws away half of the samples by sampling inside a sphere. For planar surfaces, half of the samples are deemed to lie behind the surface, having no real contribution to the ambient occlusion computation. For this reason, we prefer other methods over Crytek ambient occlusion.

5.10.2 Starcraft 2 Ambient Occlusion

Starcraft 2 ambient occlusion improves on Crytek's method by sampling inside the normal-oriented hemisphere. While the Starcraft 2 method produces excellent results, the cost of unprojecting and projecting points to and from view space inside a loop is currently too expensive for most mobile platforms. For this reason, we would prefer a method in which the cost of projection and unprojection is not linear with respect to the number of samples, but constant or zero altogether.

5.10.3 Alchemy Ambient Obscurance

Alchemy ambient obscurance is by far our favourite approach, and this is what we would implement in both applications targeting mobile and desktop. Alchemy is intuitive, efficient, simple to implement, provides artistic control via four relatively independent parameters and produces virtually no flicker with progressive mode. While not the fastest approach, Alchemy is the method that maximises both quality and performance simultaneously.

5.10.4 Horizon-Based Ambient Occlusion

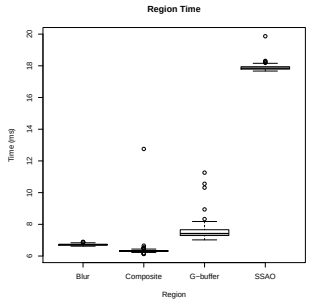
Our main issue when implementing horizon-based ambient occlusion was getting it to produce reasonable results with just eight samples. While this approach produces reasonable results with sixteen samples (a 4×4 sampling pattern), getting it to work with fewer samples posed as a challenge. In the end, we wrote an implementation of the algorithm that produces reasonable results, but the shader is too expensive to be run on mobile.

5.10.5 Home-Brewed Ambient Occlusion

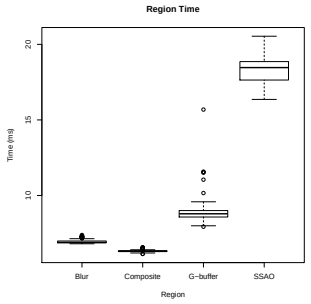
Home-brewed ambient occlusion is fast and simple to implement. However, this approach is not a true ambient occlusion method. The home-brewed approach produces edge detection and gives the scene an overall artistic look, which may or may not be desirable depending on the application. We would implement home-brewed ambient occlusion only where this artistic effect is wanted. In addition, this method is the fastest from our results, so it is indeed one we would not automatically discard.

5.10.6 Unsharp Masking of the Depth Buffer

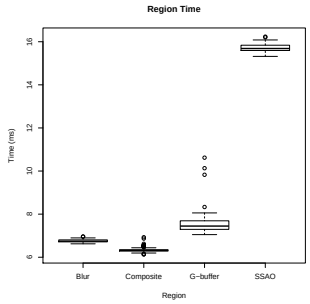
This method originally seemed promising. The method samples a rectangle centred at every pixel, so unlike random sampling, texture fetches are coherent. In addition, since this method does not rely on random sampling, no blurring of the ambient occlusion is needed to remove noise. However, we found the rectangle kernel to be too expensive on mobile platforms. For this method to produce relatively far-field effects, a large area must be sampled, increasing the runtime complexity of the shader. Otherwise, the method produces an effect similar to edge detection, which is unacceptable if ambient occlusion is what we are after.



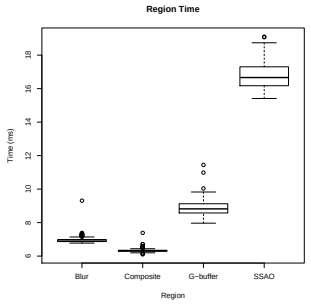
(A) Alchemy (depth).



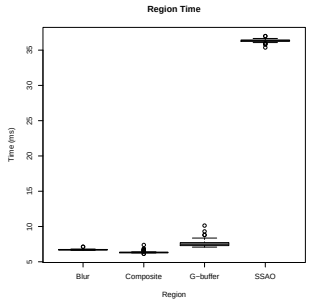
(B) Alchemy (position).



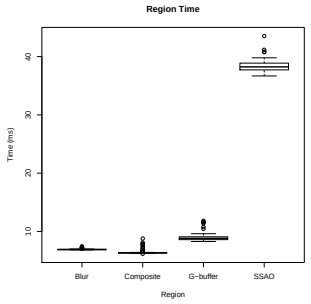
(C) Crytek (depth).



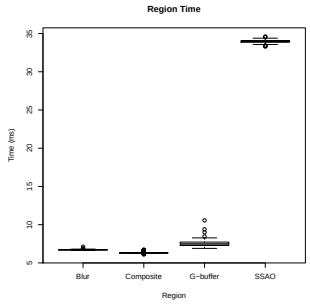
(D) Crytek (position).



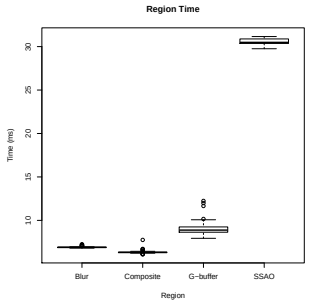
(E) HBAO (depth).



(F) HBAO (position).

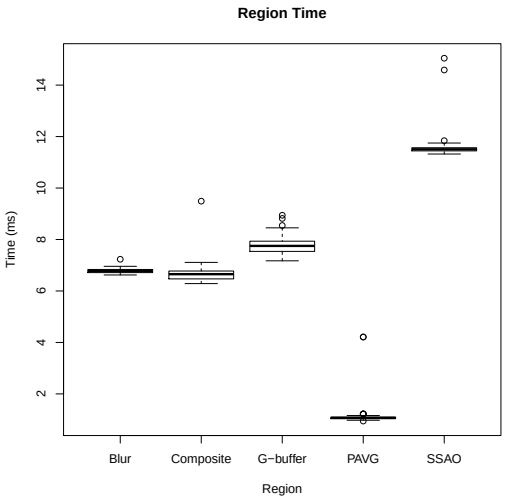


(G) SC2 (depth).

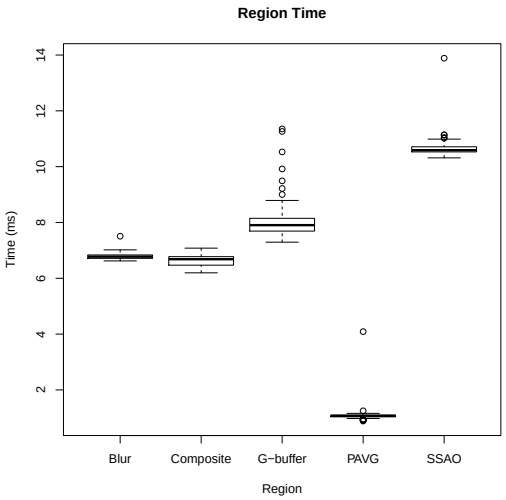


(H) SC2 (position).

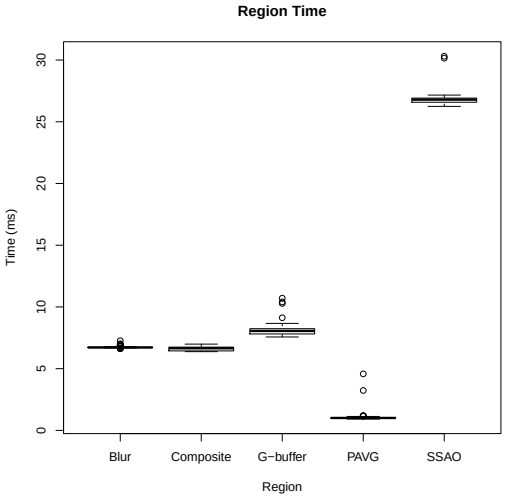
FIGURE 5.8: Profiling of each of the ambient occlusion methods using depth and position buffers.



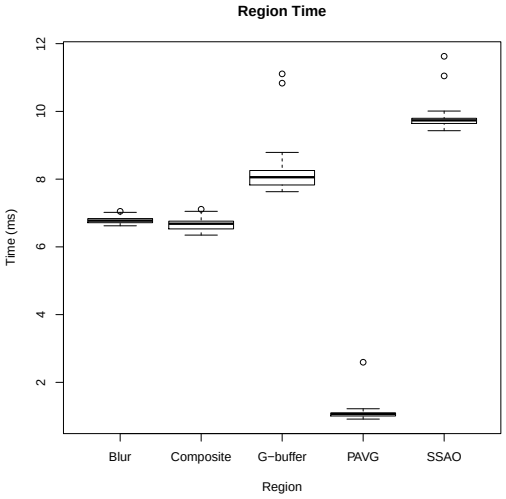
(A) Alchemy.



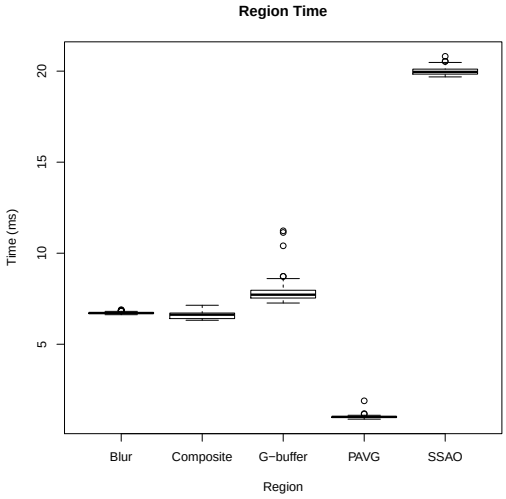
(B) Crytek.



(C) HBAO.



(D) Home-Brewed.



(E) Starcraft 2.

FIGURE 5.20: Performance of ambient occlusion methods with the progressive approach. Averaging the two partial ambient occlusion computation has a minimal impact on performance in all algorithms.

Chapter 6

Conclusions

Ambient occlusion is an approximation to global illumination that shades points as a function of their visibility, with occluded points appearing darker than non-occluded ones. While ambient occlusion is a rather crude simplification of the rendering equation, its computation in real-time is prohibitive. Instead, approximations to true ambient occlusion are typically used in real-time computer graphics.

One such approximation is screen space ambient occlusion. The main observation behind this technique is that a pair of depth and normal buffers provide an albeit crude, but nevertheless useful approximation to the 3D geometry of a scene. Screen space ambient occlusion methods use a pair of depth and normal buffers to approximate the ambient occlusion at every pixel of the screen. As a consequence, these set of techniques are relatively inexpensive to compute and suitable for real-time computer graphics applications.

The popularity of screen space ambient occlusion in real-time computer graphics applications such as PC games has only increased in the recent years. Its affordable computation cost, scalability and the fact that it can be combined with other global illumination techniques make it an excellent choice for these applications.

Unfortunately, the use of global illumination techniques in mobile games and other mobile graphics applications has traditionally been very limited. On mobile, illumination is usually baked for performance reasons, since many of these devices are still not capable of running global illumination techniques at reasonable frame rates.

While still behind desktop GPUs, mobile GPUs are evolving very rapidly. GPUs such as the Adreno 400 and 500 series or NVIDIA's K1 offer a considerable increase in computer power with respect to previous generation GPUs, and the trend is only going higher.

In this work, we study the implementation of screen space ambient occlusion on mobile. We implement several of the most popular techniques and evaluate their performance on multiple devices. In addition, we develop two rendering pipelines to support these techniques and that complement each other in terms of trade-offs. We profile and study each of the stages of these two pipelines, justifying our design decisions along

the way. In addition, we propose a screen space ambient occlusion method that is computationally inexpensive and relatively simple to implement. Finally, we propose a modification that can be applied to any screen space ambient occlusion method. This modification boosts the performance of screen space ambient occlusion methods by computing only a fraction of the occlusion in a given frame and progressively refining it in subsequent frames.

From our results, we see how screen space ambient occlusion can be computed in real-time in middle end devices such as Google's Nexus 7 (2013) and at reasonable frame rates (30-40 fps) using our pipeline and optimisations. On higher end devices such as Google's Nexus 5 or NVIDIA's Shield K1, many of the algorithms run at 60+ fps, saving part of the computational budget and leaving room for other effects.

In conclusion, we think screen space ambient occlusion is indeed possible and feasible on recent mobile devices such as Google's Nexus 5 or NVIDIA's Shield K1. Judging by the trends and the rapid evolution of mobile GPUs, we think it is only a matter of time that screen space ambient occlusion and other, more complex global illumination techniques become standard in the mobile space.

6.1 Future Work

Our work is time-bound, and we have consequently not been able to experiment as much as we initially wished. In the future, we would like to improve on our work by developing several areas of potential research.

One such area is the development of non-physically based approximations to ambient occlusion. We believe such approximations could be designed with texture access coherency in mind to provide fast approximations of ambient occlusion real-time.

Similarly, we would like to gain further insight into the implications of incoherent texture access on mobile platforms, and continue development on sampling strategies and sampling patterns to optimise memory access patterns. Given that mobile devices offer very limited memory bandwidth compared to desktop GPUs, we believe additional performance can be obtained by carefully designing such sampling strategies.

With the introduction of OpenGL ES 3.1, mobile devices supporting this version of the standard are now capable of running compute shaders. Compute shaders make the implementation of volume-based ambient occlusion approaches relatively simple, and this is another area of research we would like to explore. A volume-based approach would be decoupled from the high screen resolution offered by mobile devices, and relatively decoupled from scene complexity. In addition, no G-buffer should be needed for this set of techniques, and the illumination could be simply computed in a forward pass after using the volume to compute the ambient occlusion of the scene.

Appendix A

Ambient Occlusion Shaders

Crytek

```
// Header defined in client code
//
//#version 300 es
//#define NSAMPLES 8
//#define NSAMPLESf 8.0

#define DEPTH(p) (texture(Depth, p).r)

precision highp float;

const float Sigma = 1.1;

uniform sampler2D Depth;
uniform sampler2D Rotation;

uniform mat4 Projection;
uniform vec3 Samples[NSAMPLES];
uniform float Radius;
uniform float RotationWidth;
uniform float Far;
uniform float Near;
uniform vec2 RightTop;

in vec2 Texcoord;

layout (location = 0) out float AO;

vec3 unproject (vec2 st, float d)
{
   st = st*2.0 - 1.0;
   vec2 pnear = st * RightTop;
   float pz = -d*Far;
   return vec3(-pz*pnear.x / Near, -pz*pnear.y / Near, pz);
```

```
}
vec3 unproject (vec2 st) { return unproject(st, DEPTH(st)); }

vec3 project (vec3 p)
{
   vec4 proj = Projection * vec4(p,1.0); // coords in [-w,w]
   vec3 ndc = proj.xyz/proj.w; // ndc coords in [-1,1]
   return vec3(ndc.xy*0.5 + 0.5, -p.z/Far); // map from [-1,1] to
      [0,1]
}

// p = occludee
// q = occludder
void main ()
{
   // Occludee data
   vec2 pfrag = Texcoord; // frag in [0,1]
   float pdepth = DEPTH(pfrag);
   vec3 pview = unproject(pfrag, pdepth);

   vec2 rotst = gl_FragCoord.xy / RotationWidth;
   vec3 rvec = texture(Rotation, rotst).xyz*2.0 - 1.0;

   // Compute AO factor.

   AO = 0.0;
   for (int i = 0; i < NSAMPLES; ++i)
   {
      // Get a random sample and rotate it using the random rotation
         vector for this fragment.
      vec3 sampleVec = reflect(Samples[i], rvec);
      vec3 sampleView = pview + sampleVec;

      // Compute q in view space and vector from p to q.
      vec3 projectedSample = project(sampleView);
      vec3 qview = unproject(projectedSample.st);
      float qdepth = DEPTH(projectedSample.st);
      vec3 v = qview - pview;

      // Range check.
      // Disabling range check results in edge detection, but gets
         rid of white halos and gives
      // the AO an overall better (and more artistic) look. It also
         results in less noticeable
      // artifacts on the floor.
      //float w = abs(pview.z - qview.z) < Radius ? 1.0 : 0.0;
```

```
      AO += step(qdepth, projectedSample.z);
   }
   AO = 1.0 - Sigma*AO/NSAMPLESf;
   if (pdepth > 0.99) AO = 1.0;

   // Half of the samples are occluded, so AO mostly lies in [0,
      0.5] except for edges.
   // Remap to [0,1].
   AO = AO*2.0; // now remap to [0,1]
}
```

Starcraft 2

```
// Header defined in client code
//
//#version 300 es
//#define NSAMPLES 8
//#define NSAMPLESf 8.0

#define DEPTH(p) (texture(Depth, p).r)
#define NORMAL(p) (texture(Normal, p).rgb*2.0 - 1.0)

precision highp float;

const float Sigma = 0.8;
const float Bias = 0.35;

uniform sampler2D Depth;
uniform sampler2D Normal;
uniform sampler2D Rotation;

uniform mat4 Projection;
uniform mat4 IProjection;
uniform vec3 Samples[NSAMPLES];
uniform float Radius;
uniform float RotationWidth;
uniform float Near;
uniform float Far;
uniform vec2 RightTop;

in vec2 Texcoord;

layout (location = 0) out float AO;

vec3 unproject (vec2 st, float d)
{
```

```
   st = st*2.0 - 1.0;
   vec2 pnear = st * RightTop;
   float pz = -d*Far;
   return vec3(-pz*pnear.x / Near, -pz*pnear.y / Near, pz);
}
vec3 unproject (vec2 st) { return unproject(st, DEPTH(st)); }

vec3 project (vec3 p)
{
   vec4 proj = Projection * vec4(p,1.0); // coords in [-w,w]
   vec3 ndc = proj.xyz/proj.w; // ndc coords in [-1,1]
   return vec3(ndc.xy*0.5 + 0.5, -p.z/Far); // map from [-1,1] to
      [0,1]
}

// p = occludee
// q = occludder
void main ()
{
   // Occludee data
   vec2 pfrag = Texcoord; // frag in [0,1]
   float pdepth = DEPTH(pfrag);
   vec3 pnormal = NORMAL(pfrag);
   vec3 pview = unproject(pfrag, pdepth); // fragment position in
      view space.

   // TBN matrix
   vec2 rotst = gl_FragCoord.xy / RotationWidth;
   vec3 rvec  = texture(Rotation, rotst).xyz*2.0 - 1.0;
   vec3 tangent = normalize(rvec - pnormal * dot(rvec, pnormal));
   vec3 bitangent = cross(pnormal, tangent);
   mat3 tbn   = mat3(tangent, bitangent, pnormal);

   // Compute AO factor.

   AO = 0.0;
   for (int i = 0; i < NSAMPLES; ++i) {
      // Get a random sample and rotate it using the random rotation
         vector for this fragment.
      vec3 sampleVec = tbn * Samples[i];
      vec3 sampleView = pview + sampleVec;

      // Compute q in view space and vector from p to q.
      vec3 projectedSample = project(sampleView);
      float qdepth = DEPTH(projectedSample.st);
      vec3 qview = unproject(projectedSample.st, qdepth);
      vec3 v = qview - pview;
```

```
      // Range check.
      float w = abs(pview.z - qview.z) - Bias < Radius ? 1.0 : 0.0;

      // Use cos(angle) between sample point and q as a weight.
      w *= max(0.0, dot(normalize(v), pnormal));

      AO += w * step(qdepth, projectedSample.z);
   }
   AO = 1.0 - Sigma*AO/NSAMPLESf;
   if (pdepth > 0.99) AO = 1.0; // Do not occlude background
}
```

Alchemy

```
// Header defined in client code
//
//#version 300 es
//#define NSAMPLES 8
//#define NSAMPLESf 8.0

#define DEPTH(p) (texture(Depth, p).r)
#define NORMAL(p) (texture(Normal, p).rgb*2.0 - 1.0)

precision highp float;

const float Beta = 0.005;
const float Eps = 0.003;
const float Sigma = 0.09;

//uniform sampler2D Depth;
uniform sampler2D Depth;
uniform sampler2D Normal;
uniform sampler2D Rotation;

uniform mat4 IProjection;
// Samples along ray.
// Samples are scaled by the view space hemisphere radius and the
   texture space
// scaling factor. The samples need only be divided by view space z
   to obtain the
// final vector length.
uniform vec2 Samples[NSAMPLES];
uniform float RotationWidth;
uniform float Far;
uniform float Near;
```

```
uniform vec2 RightTop;

in vec2 Texcoord;

layout (location = 0) out float AO;

// Unproject the given point in texture coordinates to view space
    coordinates.
// (s,t, linear depth) -> (x,y,z)
vec3 unproject (vec2 st, float d)
{
   st = st*2.0 - 1.0;
   vec2 pnear = st * RightTop;
   float pz = -d*Far;
   return vec3(-pz*pnear.x / Near, -pz*pnear.y / Near, pz);
}
vec3 unproject (vec2 st) { return unproject(st, DEPTH(st)); }

void main ()
{
   vec2 pfrag = Texcoord; // frag in [0,1]
   float pdepth = DEPTH(pfrag);
   vec3 pnormal = NORMAL(pfrag);
   vec3 pview = unproject(pfrag, pdepth);

   vec2 rotst = gl_FragCoord.xy / RotationWidth;
   vec2 rvec = texture(Rotation, rotst).xy*2.0 - 1.0;

   AO = 0.0;
   for (int i = 0; i < NSAMPLES; ++i)
   {
      vec2 svec = reflect(Samples[i], rvec) / -pview.z;
      vec2 qfrag = pfrag + svec;
      vec3 qview = unproject(qfrag);

      vec3 v = qview - pview;

      AO += max(0.0, (dot(v,pnormal) - Beta) / (dot(v,v) + Eps));
   }
   AO = max(0.0, 1.0 - 2.0*Sigma/NSAMPLESf*AO);
   if (pdepth > 0.99) AO = 1.0; // Do not occlude background
}
```

HBAO

```
// Header defined in client code
```

```
//
//#version 300 es
//#define N_SAMPLES_PER_RAY 2
//#define N_SAMPLES_PER_RAYf 2.0

#define N_RAYS 4
#define N_RAYSf 4.0

#define DEPTH(p) (texture(Depth, p).r)

#define RAD 0.01745329251

precision highp float;

const float Bias = 35.0*RAD;
const float Sigma = 2.0;

uniform sampler2D Depth;
uniform sampler2D Rotation;

// Samples along ray.
// Samples are scaled by the view space hemisphere radius and the
   texture space
// scaling factor. The samples need only be divided by view space z
   to obtain the
// final vector length.
//
// Samples are also jittered using a normal distribution to reduce
   artifacts.
uniform vec2 RaymarchDir[N_RAYS];
uniform float Samples[N_SAMPLES_PER_RAY];
uniform float RadiusSquared; // hemisphere radius squared
uniform float RotationWidth;
uniform float Near;
uniform float Far;
uniform vec2 RightTop;

in vec2 Texcoord;

layout (location = 0) out float AO;

vec3 unproject (vec2 st, float d)
{
   st = st*2.0 - 1.0;
   vec2 pnear = st * RightTop;
   float pz = -d*Far;
   return vec3(-pz*pnear.x / Near, -pz*pnear.y / Near, pz);
```

```
}
vec3 unproject (vec2 st) { return unproject(st, DEPTH(st)); }

// Return sin(x) given tan(x)
float tan2sin (float tan_x)
{
   return tan_x * pow(tan_x*tan_x + 1.0, -0.5);
}

// p = occludee
// q = occludder
void main ()
{
   vec2 pfrag = Texcoord; // frag in [0,1]
   vec3 pview = unproject(pfrag);

   // compute random rotation
   vec2 rotst = gl_FragCoord.xy / RotationWidth;
   vec2 rvec = texture(Rotation, rotst).xy*2.0 - 1.0;
   mat2 TN = mat2(rvec, vec2(-rvec.y, rvec.x));

   // compute derivatives to later find the tangent vector
   vec2 depth_texel_size = vec2(1.0) / vec2(textureSize(Depth,0));

   AO = 0.0;
   for (int i = 0; i < N_RAYS; ++i)
   {
      vec2 dir  = TN*RaymarchDir[i];
      vec3 pright = unproject(pfrag + dir*depth_texel_size);
      vec3 pleft = unproject(pfrag - dir*depth_texel_size);
      vec3 tangent = pright-pleft; // no need to normalise

      // initialise the maximum seen horizon angle to the tangent
         angle
      float tan_max_horizon = tangent.z / length(tangent.xy) +
         tan(Bias);
      float sin_max_horizon = tan2sin(tan_max_horizon);

      // raymarch depth buffer
      for (int j = 0; j < N_SAMPLES_PER_RAY; ++j)
      {
         vec2 qfrag = pfrag + Samples[j]*dir / -pview.z;
         vec3 qview = unproject(qfrag);

         vec3 horizon = qview - pview;
         float horizon_length_squared = dot(horizon,horizon);
         float tan_horizon = horizon.z / length(horizon.xy);
```

```
            // if K <= 1, then the sample is within the hemisphere
               radius
            float K = horizon_length_squared / RadiusSquared;

            if (K <= 1.0 && tan_horizon > tan_max_horizon)
            {
               float sin_horizon =
                  tan2sin(tan_horizon);//*sign(Samples[j]);
               AO += (sin_horizon - sin_max_horizon) * (1.0 - K);
               tan_max_horizon = tan_horizon;
               sin_max_horizon = sin_horizon;
            }
         }
      }
      AO = 1.0 - Sigma*AO/(N_RAYSf);
}
```

Home-Brewed

```
// Header defined in client code
//
//#version 300 es
//#define NSAMPLES 8
//#define NSAMPLESf 8.0

#define DEPTH(p) (texture(Depth, p).r)
#define NORMAL(p) (texture(Normal, p).rgb*2.0 - 1.0)

precision highp float;

uniform sampler2D Depth;
uniform sampler2D Normal;
uniform sampler2D Rotation;

uniform vec2 Samples[NSAMPLES];
uniform float RotationWidth;

in vec2 Texcoord;

layout (location = 0) out float AO;

void main ()
{
   vec2 pfrag = Texcoord; // fragment position in [0,1]
   float pdepth = DEPTH(pfrag);
```

```
   vec3 pnormal = NORMAL(pfrag);

   vec2 rotst = gl_FragCoord.xy / RotationWidth;
   vec2 rvec = texture(Rotation, rotst).xy*2.0 - 1.0;

   AO = 0.0;
   for (int i = 0; i < NSAMPLES; ++i)
   {
      vec2 qfrag = pfrag + reflect(Samples[i].xy, rvec) / pdepth;
      float qdepth = DEPTH(qfrag);
      vec3 qnormal = NORMAL(qfrag);

      float diff = max(0.0, qdepth - pdepth);

      // Avoid self-shadowing
      float w = 1.0 - dot(pnormal, qnormal);

      // Penalise large depth discontinuities
      float odiff = 1.0 + diff;
      w *= smoothstep(0.0, 1.0, odiff*odiff);

      AO += w * (1.0 - diff);
   }
   AO = 1.0 - AO/NSAMPLESf;
}
```

Unsharp Mask

```
#version 300 es

precision highp float;

uniform sampler2D Input;

uniform vec2 TexelSize;
uniform float KernelSize;

uniform float A;
uniform float K;

out float Output;

in vec2 Texcoord;

void main ()
{
```

```
   float Blur = 0.0;
   float W = 0.0;
   for (float x = -KernelSize; x <= KernelSize; x += 1.0)
   {
      for (float y = -KernelSize; y <= KernelSize; y += 1.0)
      {
         vec2 blurSample = Texcoord + vec2(x,y) * TexelSize;

         vec2 v = blurSample - Texcoord;
         float w = A * exp(-dot(v,v) * K);

         Blur += texture(Input, blurSample).r * w;
         W += w;
      }
   }
   Blur = Blur / W;

   float In = texture(Input, Texcoord).r;
   Output = (In - Blur)*0.5 + 0.5;
};
```

Appendix B

Blur Filters

Bilateral Filter

```
#version 300 es

#define DEPTH(p) (texture(Depth,p).r)

precision highp float;

uniform sampler2D Input;
uniform sampler2D Depth;

uniform vec2 TexelSize;
uniform float KernelSize;
uniform float Far;

// pre-computed gaussian filter constants
uniform float A;
uniform float B;
uniform float K;

out float Output;

in vec2 Texcoord;

void main ()
{
   float pdepth = DEPTH(Texcoord);

   Output = 0.0;
   float W = 0.0;
   for (float x = -KernelSize; x <= KernelSize; x += 1.0)
   {
      for (float y = -KernelSize; y <= KernelSize; y += 1.0)
      {
         vec2 blurSample = Texcoord + vec2(x,y) * TexelSize;
```

```
            // space weight
            vec2 v = blurSample - Texcoord;
            float w = A * exp(-dot(v,v) * K);

            // range weight
            float qdepth = DEPTH(blurSample);
            w *= A * exp(-abs(pdepth - qdepth) * B);

            Output += texture(Input, blurSample).r * w;
            W += w;
        }
    }
    Output = Output / W;
}
```

Separable Blur

```
#version 300 es

#define DEPTH(p) (texture(Depth,p).r)

precision highp float;

uniform sampler2D Input;
uniform sampler2D Depth;

uniform float TexelSize; // texel size across blur direction (x or y)
uniform float KernelSize;
uniform float Far;
uniform vec2 Dir; // blur direction

// pre-computed gaussian filter constants
uniform float A;
uniform float B;
uniform float K;

out float Output;

in vec2 Texcoord;

void main ()
{
    float pdepth = DEPTH(Texcoord);

    Output = 0.0;
```

```
    float W = 0.0;
    for (float k = -KernelSize; k <= KernelSize; k += 1.0)
    {
        vec2 blurSample = Texcoord + k*Dir * TexelSize;

        // space weight
        vec2 v = blurSample - Texcoord;
        float w = A * exp(-dot(v,v) * K);

        // range weight
        float qdepth = DEPTH(blurSample);
        w *= A * exp(-abs(pdepth - qdepth) * B);

        Output += texture(Input, blurSample).r * w;
        W += w;
    }
    Output = Output / W;
}
```

Bibliography

[Aal13] Frederik Peter Aalund. "A Comparative Study of Screen-Space Ambient Occlusion Methods". 2013. URL: http://frederikaalund.com/a-comparative-study-of-screen-space-ambient-occlusion-methods/.

[Amd] "High Bandwidth Memory | Reinventing Memory Technology". In: ().

[Bri07] Robert Bridson. "Fast Poisson Disk Sampling in Arbitrary Dimensions". In: *ACM SIGGRAPH 2007 Sketches*. SIGGRAPH '07. San Diego, California: ACM, 2007. DOI: 10.1145/1278780.1278807. URL: http://doi.acm.org/10.1145/1278780.1278807.

[BSD08] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. "Image-space Horizon-based Ambient Occlusion". In: *ACM SIGGRAPH 2008 Talks*. SIGGRAPH '08. Los Angeles, California: ACM, 2008, 22:1–22:1. ISBN: 978-1-60558-343-3. DOI: 10.1145/1401032.1401061. URL: http://doi.acm.org/10.1145/1401032.1401061.

[Cha11] John Chapman. "SSAO Tutorial". In: (2011).

[Fel15] Mark Feldman. "Adreno Rendering Tutorial 1: Choosing Resolution". In: (2015). URL: https://developer.qualcomm.com/software/adreno-gpu-sdk/tutorial-videos.

[FM08] Dominic Filion and Rob McNaughton. "Effects & Techniques". In: *ACM SIGGRAPH 2008 Games*. SIGGRAPH '08. Los Angeles, California: ACM, 2008, pp. 133–164. DOI: 10.1145/1404435.1404441. URL: http://doi.acm.org/10.1145/1404435.1404441.

[Fol14] Denis Foley. "NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data". In: (2014).

[Gre09] Simon Green. "NVIDIA Effects, GDC 2009". In: (2009). URL: http://www.slideshare.net/IGDA_London/nvidia-effects-gdc09.

[Hua+11] Jing Huang et al. "Separable Approximation of Ambient Occlusion". In: *Eurographics 2011 - Short papers*. 2011.

[Ios] "OpenGL ES Programming Guide for iOS". In: (). URL: https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html.

[KL05a] Janne Kontkanen and Samuli Laine. "Ambient Occlusion Fields". In: *Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games*. ACM Press, 2005, pp. 41–48.

[KL05b] Janne Kontkanen and Samuli Laine. "Ambient Occlusion Fields". In: *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. I3D '05. Washington, District of Columbia: ACM, 2005, pp. 41–48. ISBN: 1-59593-013-2. DOI: 10.1145/1053427.1053434. URL: http://doi.acm.org/10.1145/1053427.1053434.

[KS11] Brian Klug and Anand Lal Shimpi. "Understanding Rendering Techniques". In: (2011). URL: http://www.anandtech.com/show/4686/samsung-galaxy-s-2-international-review-the-best-redefined/15.

[LCD06] Thomas Luft, Carsten Colditz, and Oliver Deussen. "Image Enhancement by Unsharp Masking the Depth Buffer". In: *ACM SIGGRAPH 2006 Papers*. SIGGRAPH '06. Boston, Massachusetts: ACM, 2006, pp. 1206–1213. ISBN: 1-59593-364-6. DOI: 10.1145/1179352.1142016. URL: http://doi.acm.org/10.1145/1179352.1142016.

[McG+11] Morgan McGuire et al. "The Alchemy Screen-Space Ambient Obscurance Algorithm". In: *High-Performance Graphics 2011*. Vancouver, BC, Canada, 2011. URL: http://graphics.cs.williams.edu/papers/AlchemyHPG11/.

[McG10] Morgan McGuire. "Ambient Occlusion Volumes". In: *Proceedings of High Performance Graphics 2010*. Saarbrucken, Germany, 2010. URL: http://graphics.cs.williams.edu/papers/AOVHPG10.

[Mer12] Bruce Merry. "Performance Tuning for Tile-Based Architectures". In: *OpenGL Insights*. Ed. by Patrick Cozzi and Christophe Riccio. http://www.openglinsights.com/. CRC Press, 2012, pp. 323–335. ISBN: 978-1439893760.

[Mit07] Martin Mittring. "Finding Next Gen: CryEngine 2". In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH '07. San Diego, California: ACM, 2007, pp. 97–121. ISBN: 978-1-4503-1823-5. DOI: 10.1145/1281500.1281671. URL: http://doi.acm.org/10.1145/1281500.1281671.

[Mit12] Martin Mittring. "The Technology Behind the Unreal Engine 4 Elemental demo". In: 2012.

[MML12] Morgan McGuire, Michael Mara, and David Luebke. "Scalable Ambient Obscurance". In: *High-Performance Graphics 2012*. Paris, France, 2012. URL: http://graphics.cs.williams.edu/papers/SAOHPG12/.

[MSW10] Oliver Mattausch, Daniel Scherzer, and Michael Wimmer. "High-Quality Screen-Space Ambient Occlusion using Temporal Coherence". In: *Computer Graphics Forum* 29.8 (Dec. 2010), pp. 2492–2503. ISSN: 0167-7055. URL: http://www.cg.tuwien.ac.at/research/publications/2010/mattausch-2010-tao/.

[Ori15] Ayo Orimoloye. "Adreno Hardware Tutorial 3: Tile Based Rendering". In: (2015). URL: https://developer.qualcomm.com/software/adreno-gpu-sdk/tutorial-videos.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005. ISBN: 0321335597.

[PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123750792, 9780123750792.

[PV05] Tuan Q. Pham and Lucas J. van Vliet. "Separable bilateral filtering for fast video preprocessing." In: *ICME*. IEEE Computer Society, 2005, pp. 454–457. ISBN: 0-7803-9331-7. URL: http://dblp.uni-trier.de/db/conf/icmcs/icme2005.html#PhamV05.

[QT15] Inc. Qualcomm Technologies. "Qualcomm Adreno OpenGL ES Developer Guide". In: (2015). URL: https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html.

[Rit+08] Tobias Ritschel et al. "3D Unsharp Masking for Scene Coherent Enhancement". In: *ACM Trans. Graph. (Proc. of SIGGRAPH 2008)* 27.3 (2008).

[SA07] Perumaal Shanmugam and Okan Arikan. "Hardware Accelerated Ambient Occlusion Techniques on GPUs". In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. I3D '07. Seattle, Washington: ACM, 2007, pp. 73–80. ISBN: 978-1-59593-628-8. DOI: 10.1145/1230100.1230113. URL: http://doi.acm.org/10.1145/1230100.1230113.

[Som15] Rys Sommefeldt. "A Look at the PowerVR Graphics Architecture: Tile-Based Rendering". In: (2015). URL: http://blog.imgtec.com/powervr/a-look-at-the-powervr-graphics-architecture-tile-based-rendering.

[SPD07] Jack Tumblin Sylvain Paris Pierre Kornprobst and Frédo Durand. "A Gentle Introduction to Bilateral Filtering and its Applications". In: (2007).

[Suf] .

[Tim13] Ville Timonen. "Line-Sweep Ambient Obscurance". In: *Computer Graphics Forum (Proceedings of EGSR 2013)* 32.4 (2013), pp. 97–105. URL: http://wili.cc/research/lsao/.

[TL06] Oliver Deussen Thomas Luft Carsten Colditz. "Image Enhancement by Unsharp Masking the Depth Buffer". In: (2006).